



École Polytechnique de l'Université de Tours  
64, Avenue Jean Portalis  
37200 TOURS, FRANCE  
Tél. +33 (0)2 47 36 14 14  
[www.polytech.univ-tours.fr](http://www.polytech.univ-tours.fr)

## Département Informatique

### Génie logiciel

# Convention de nommage

#### Superviseurs

Emmanuel Néron  
[emmanuel.neron@univ-tours.fr](mailto:emmanuel.neron@univ-tours.fr)  
Nicolas Ragot  
[nicolas.ragot@univ-tours.fr](mailto:nicolas.ragot@univ-tours.fr)

Université François Rabelais, Tours

#### Étudiant

Mathieu de Brito  
[mathieu.debrito@etu.univ-tours.fr](mailto:mathieu.debrito@etu.univ-tours.fr)

DI5 2009 - 2010

Version du 12 octobre 2010

# Table des matières

---

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Définition . . . . .	4
1.2	Avantages . . . . .	4
1.3	Dangers . . . . .	4
<b>2</b>	<b>Nommage</b>	<b>5</b>
2.1	Classes . . . . .	5
2.2	Méthodes . . . . .	5
2.3	Constantes . . . . .	6
2.4	Variables . . . . .	6
2.5	Fichiers . . . . .	6
<b>3</b>	<b>Commentaires</b>	<b>7</b>
3.1	Utilisation . . . . .	7
3.2	Implémentation . . . . .	7
3.2.1	Bloc . . . . .	7
3.2.2	Précédence . . . . .	7
3.2.3	Fin de ligne . . . . .	8
3.3	Documentation . . . . .	8
3.3.1	Fichier . . . . .	8
3.3.2	Classe . . . . .	8
3.3.3	Méthode . . . . .	9
3.3.4	Attribut . . . . .	9
<b>4</b>	<b>Déclarations</b>	<b>10</b>
4.1	Visibilité et accès aux données . . . . .	10
4.2	Classes & Méthodes . . . . .	10
4.3	Attributs . . . . .	11
4.4	Variables . . . . .	11
<b>5</b>	<b>Style &amp; lisibilité</b>	<b>13</b>
5.1	Caractères par ligne . . . . .	13
5.2	Encodage . . . . .	13
5.3	Indentation . . . . .	13
5.4	Méthodes . . . . .	14
5.5	Parenthèses . . . . .	14
5.6	Accolades . . . . .	14
5.7	Espaces . . . . .	15
<b>6</b>	<b>Structures de contrôle</b>	<b>16</b>
6.1	Instructions . . . . .	16
6.2	Retour de fonction . . . . .	16
6.3	Si-alors-sinon . . . . .	17

6.4	Tant que . . . . .	17
6.5	Pour . . . . .	17
6.6	Pour chaque . . . . .	18
6.7	Répéter-Jusqu'à . . . . .	18
6.8	Switch . . . . .	18
6.9	Break & continue . . . . .	19
6.10	Opérateur ternaire . . . . .	19
<b>7</b>	<b>Débogage</b>	<b>21</b>
7.1	Benchmark . . . . .	21
7.1.1	Réutilisation . . . . .	21
<b>8</b>	<b>Organisation des sources</b>	<b>22</b>
<b>A</b>	<b>CheckStyle</b>	<b>23</b>
A.1	Installation . . . . .	23
<b>B</b>	<b>Javadoc et Doxygen</b>	<b>24</b>
B.1	Javadoc . . . . .	24
B.2	Doxygen . . . . .	24
<b>C</b>	<b>Quelques illustrations</b>	<b>26</b>

# Introduction

---

## 1.1 Définition

Une convention, ce sont des procédures, des méthodes, des langages imposés ou préconisés par les normes adaptées à l'environnement d'utilisation afin de favoriser la production et la maintenance de composants logiciels de qualité. (source : Wikipédia [8])

En d'autres termes, une convention est une approche des bonnes pratiques de programmation qui permet à tout développeur de coder proprement.

Une convention n'est pas l'unique vérité sur la manière de bien programmer. En revanche, elle peut offrir une approche, un style de programmation dont la mise œuvre est simple et la compréhension du code généré à l'aide de cette convention est compréhensible par tous.

## 1.2 Avantages

Les raisons pour lesquelles on utilise une convention de nommage (par opposition à l'autorisation accordée aux programmeurs de choisir n'importe quel style de nommage) sont les suivantes (source : Wikipédia [6]) :

- sécuriser l'application par le respect de règles précises ;
- promouvoir la cohérence dans une équipe de développement ;
- faciliter une éventuelle maintenance ;
- améliorer l'apparence esthétique et professionnelle du produit.

## 1.3 Dangers

Le choix des conventions de nommage fait souvent l'objet de débats. En effet, l'idée étant de guider la manière de programmer à des personnes ayant leurs propres habitudes, il est possible que tout ajustement puisse semer la confusion.

Lorsque l'on crée des règles il faut être minutieux car si elles sont incohérentes, arbitraires, difficiles à mémoriser ou plus fastidieuses que bénéfiques, le défi d'unifier les bonnes pratiques de programmation se révélera inutile.

# Nommage

---

Le nommage des différents éléments d'un programme est crucial pour la compréhension du code. Un nom est le résultat d'un long processus de pensée résultant du sujet étudié. Si les noms sont appropriés alors tout les éléments s'adaptent ensembles naturellement. De plus, les programmeurs pourront reprendre le code et le comprendre aisément.

L'utilisation de la langue de Shakespeare est conseillée mais le choix est laissé au développeur. Une fois la décision prise, le développeur doit s'y tenir.

Voici un ensemble de recommandations valables pour la plupart des langages usuels.

## 2.1 Classes

Il est d'usage que les noms de classes soient constitués de mots, en l'occurrence, de noms communs. Chaque mot commence par une majuscule, le reste étant en minuscule. Les mots sont concaténés sans "\_" Les abréviations sont à éviter pour faciliter la compréhension (sauf cas communs : XML, URL, etc.).

**Exemple :**

- classe Voiture
- classe ParseurXML

## 2.2 Méthodes

Habituellement chaque méthode effectue une action, ainsi le nom doit être suffisamment explicite pour savoir ce nom qu'elle fait. les règles minuscules/majuscules sont les mêmes que pour les classes, sauf pour le premier mot qui commence par une minuscule.

**Exemple :**

- manger()
- avancerVoiture()

Le cas des *getters* et des *setters* est un peu particulier. En effet, on utilise le préfixe *get* pour la récupération des variables ou le préfixe *is* si la variable à récupérer est booléenne. De même, on utilise le préfixe *set* pour la mise à jour des variables.

**Exemple :**

- getNomVariable() /\* Récupération de la variable nomVariable \*/
- isNomVariable() /\* Récupération de la variable booléenne nomVariable \*/
- setNomVariable([...]) /\* Mise à jour de la variable nomVariable \*/

## 2.3 Constantes

La plupart des normes utilisent uniquement les majuscules pour les noms de constantes, chaque mot étant séparé par des "\_".

**Exemple :**

- HAUTEUR
- LONGUEUR\_MAX

## 2.4 Variables

Les noms de variables sont extrêmement importants. En effet, c'est ce qui va le plus aider à la compréhension du code. Ainsi chaque variable doit posséder un nom cohérent avec son utilisation. Aussi, l'utilisation du *i* dans une boucle est à proscrire sauf cas cohérent (formule mathématique, coordonnées, etc.).

**Exemple :**

- voitures // pour un objet contenant des voitures
- numVoiture // pour les boucles
- compteurDeRoues // pour les boucles (autre exemple)

## 2.5 Fichiers

Pour ce qui est des noms de fichiers, si c'est un fichier de classe, en général, le nom du fichier est le nom de la classe. Sinon, le nom décrira au mieux l'utilisation du fichier. On peut aussi prendre en compte l'arborescence (ou la hiérarchie) si cela peut faciliter la compréhension.

**Exemple :**

- FonctionsUtiles
- VueAdminVoitureAjouter //on rappelle la hiérarchie

# Commentaires

---

La documentation est un outil professionnel pour le travail collaboratif surtout mais également pour le travail « en solitaire ». En effet, les subtilités de la programmation de tout un chacun peuvent échapper aux autres. Il est donc de rigueur de laisser la possibilité à quelqu'un d'externe au projet de comprendre les fonctionnalités et la réalisation sans regarder les détails du code, voire sans le consulter du tout.

Il y a deux types de commentaires : les commentaires d'implémentation (`//`, `/* [...] */`) et les commentaires de documentation (`/** [...] */`).

## 3.1 Utilisation

Les commentaires, dans un programme, servent avant tout à :

- utiliser de façon intelligente l'auto-complétion (paramètres expliqués) ;
- expliquer une portion de code en vue de sa maintenance.

Il est conseillé d'écrire les commentaires d'un bout de code au moment même où il est écrit, avant l'oubli de ses subtilités.

## 3.2 Implémentation

Les commentaires d'implémentation s'utilisent pour décrire les actions du code auquel on fait référence. Ils sont utilisés notamment pour expliquer un bout de code particulier, décrire une spécificité. Les commentaires d'implémentation sont sous la forme `/* */` (sauf exception : commentaire de fin de ligne).

### 3.2.1 Bloc

Si un commentaire est conséquent, alors il est mis sous la forme d'un bloc, c'est-à-dire sur plusieurs lignes.

#### Exemple :

```
/*  
 * Un commentaire  
 * sur plusieurs lignes  
 */
```

### 3.2.2 Précédence

Les commentaires qui précèdent une ligne de code doivent expliquer de manière succincte l'action réalisée. Pour plus de clarté, une ligne vide est ajoutée avant le commentaire.

**Exemple :**

```
/* commentaire sur action 1 */  
[action 1]
```

```
/* commentaire sur action 2 */  
[action 2]
```

### 3.2.3 Fin de ligne

Les commentaires placés en fin de ligne sont de la forme `//`. Ils sont indentés et très courts.

**Exemple :**

```
[action 1]           // commentaire court sur action 1
```

```
[action 2]           // commentaire court sur action 2
```

## 3.3 Documentation

Les commentaires de documentation s'utilisent pour décrire brièvement et de manière générale l'utilisation du code auquel on fait référence. On essaiera de donner une vue d'ensemble de l'implémentation.

L'idée de ces commentaires est de pouvoir générer automatiquement à l'aide d'outils (Javadoc et Doxygen par exemple) la documentation d'un projet dans le but d'avoir une vision synthétique et claire de ce dernier. Ces commentaires se placent en en-tête des fichiers, des classes, des fonctions et des variables.

Il convient ici de s'adapter aux balises utilisées par l'outil de documentation choisi et de remplir les différents champs (cf. annexe [B.1](#) et [B.2](#)).

### 3.3.1 Fichier

La documentation d'un fichier se place en-tête du fichier.

**Forme :**

```
/** Description  
 * @author Nom de l'auteur  
 * @version Date de dernière modification + commentaires sur la modification  
 */
```

### 3.3.2 Classe

La documentation d'une classe se place juste avant sa déclaration.

**Forme :**

```
/** Description  
 * @author Nom de l'auteur  
 */
```

```
Classe MaClasse
```

```
...
```



### 3.3.3 Méthode

La documentation d'une méthode se place juste avant sa déclaration.

**Forme :**

```
/** Description  
 * @author Nom de l'auteur  
 * @param typeParamètre nomParamètreVisé Description  
 * @return typeRetourné Description  
 * @throws typeException Description de l'éventuel problème  
 */
```

**Fonction** typeRetourné maFonction ()

...

### 3.3.4 Attribut

La documentation d'un attribut se place juste avant sa déclaration.

**Forme :**

```
/** @var typeMaVariable Description  
 */  
visibilitéMaVariable typeMaVariable maVariable ;
```

# Déclarations

---

## 4.1 Visibilité et accès aux données

La visibilité des entités est une information à ne pas négliger. En effet, même si les compilateurs s'accordent sur la manière de gérer l'héritage par défaut, tout le monde n'est pas forcément au fait des bonnes pratiques et rappeler ou expliciter la visibilité utilisée permet de signaler que le programmeur a conscience de ce qu'il fait.

Il faut bien réfléchir à la visibilité des attributs ainsi qu'à leur accès. En effet, la notion d'objet permet d'englober les informations afin de les sécuriser : l'encapsulation des données est un des concepts essentiels de la programmation. Si l'on ne respecte pas les règles d'encapsulation, on peut permettre à l'utilisateur de modifier certains attributs n'importe comment, ce qui sera source de nombreux problèmes.

En général, on évite la mise en *public* des attributs (on force leur manipulation par l'usage de méthodes sécurisées) et on donne la visibilité *protected* aux attributs des classes pouvant être dérivées.

## 4.2 Classes & Méthodes

Les déclarations des classes et méthodes se font dans un ordre logique. Par exemple, on essaiera de classer les déclarations en les regroupant de façons thématiques (les *getters* et les *setters* ensemble, etc.) et selon leur ordre d'importance (les sous-fonctions plutôt en dernier).

### Exemple :

```
/**
 * Documentation classe
 */
classe Voiture
{
    /* Déclaration des attributs */

    /* Déclaration des constructeurs */

    /* Déclaration des méthodes statiques */

    /* Déclaration des surcharges */

    /* Déclaration des méthodes métier */

    /* Déclaration des getters & setters */
}
```

### 4.3 Attributs

La déclaration des attributs se fait dans un ordre logique. On essaiera donc de classer les déclarations selon leur ordre d'importance ou de définition dans le cadre d'une relation objet-base de données. S'appuyer sur la visibilité est également recommandé (attributs publics en premier).

**Exemple :**

```
/**
 * Documentation attribut 1 (important)
 */
visibilite typeAttribut attribut 1 ;

/**
 * Documentation attribut 2 (inutile)
 */
visibilite typeAttribut attribut 2 ;
```

### 4.4 Variables

Les déclarations de variables se font au début du bloc dans lequel elles interviennent (portée restreinte au bloc). Un commentaire de fin de ligne pour rappeler brièvement l'utilisation de cette variable peut être ajouté pour faciliter la lecture.

Pour faciliter la compréhension, une seule déclaration par ligne est autorisée. En effet, la multiplicité des déclarations de variables sur une ligne peut conduire à des confusions et faire perdre du temps lors de la lecture.

Essayez d'initialiser vos variables à l'endroit où elles sont déclarées. C'est en effet le premier endroit que l'on va inspecter pour connaître la valeur initiale de la variable.

Pour ce qui est des variables de boucle, si la variable n'est pas utilisée en dehors de la boucle, la déclaration peut se faire à l'intérieur de la boucle (si le langage le permet), sinon la variable est déclarée comme les autres.

**Exemple :**

```
/* — KO — */
typeAttribut1 attribut 1, attribut 2, attribut 3 ;
typeAttribut2 attribut 4, attribut 5, attribut 6 ;
Bloc 1
{ // Rappeler que les variables sont visibles dans le bloc concerné
/* — OK — */
typeAttribut1 attribut 1 ; // Brève explication sur attribut 1
typeAttribut1 attribut 2 ; // Brève explication sur attribut 2
typeAttribut1 attribut 3 ; // Brève explication sur attribut 3
etc...
```

```
Bloc 2 (typeMaVariable maVariable de bloc 2; maVariable.conditionOK(); maVariable.suivant())  
{  
    [...]  
}  
} // Fin du bloc 1
```

Remarque : Il est très fortement déconseillé d'utiliser le même nom de variable que celui d'un attribut. Aussi, pensez à bien définir les noms pour qu'ils ne se confondent pas.

# Style & lisibilité

---

La plupart des IDE permettent de régler les paramètres présentés ci-dessous.

## 5.1 Caractères par ligne

Il est recommandé que la longueur des lignes ne dépasse pas 75 à 85 caractères car certains écrans ou IDE ne le gèrent pas très bien. De plus, c'est le standard d'impression A4.

## 5.2 Encodage

Il est recommandé d'utiliser un encodage multi-plateforme comme UTF-8 (exemple : problème des sauts de lignes entre Windows et Unix).

## 5.3 Indentation

L'indentation est une chose primordiale à la compréhension rapide du code mais aussi à la motivation nécessaire à sa maintenance : un code clair donne plus envie de se replonger dedans qu'un code brouillon.

Le nombre d'espaces d'une indentation est généralement de 4. En général, les tabulations sont préférées aux espaces. Dans un souci de portabilité (éditeurs différents, OS différents), ne pas mélanger les tabulations et les espaces lors de l'indentation.

Tout nouveau bloc de code doit être indenté. Le niveau de hiérarchie dans le code indique l'indentation à respecter : plus le niveau d'imbrication est important, plus l'indentation est importante.

Il est bon de savoir aussi que la plupart des IDE (netBeans, Eclipse, etc.) possèdent un formatage automatique du code (raccourcis : regarder dans les menus de l'IDE ou sur internet).

### Exemple :

```
/* nouveau bloc (hiérarchie = 1) */  
[action]
```

**Pour** (initialisation ; condition ; avancement)

```
{  
    /* nouveau bloc (hiérarchie = 2) */  
    [action]
```

**Si** (condition) **Alors**

```
{  
    /* nouveau bloc (hiérarchie = 3) */  
    [action]
```

```
}  
}
```

## 5.4 Méthodes

L'idéal pour une méthode est de ne pas dépasser 40 lignes. En effet, passé ce nombre, le re-lecteur est démotivé. De ce fait, le programmeur doit se poser la question de savoir s'il serait judicieux de créer une autre méthode pour alléger cette dernière.

Si, lors d'une déclaration ou d'un appel à une fonction, il y a trop de paramètres, alors vous pouvez revenir à la ligne après une virgule. Les retours à la ligne forcés doivent alors être indentés jusqu'au niveau de la parenthèse ouvrante de la fonction. Il est à noter que le passage de beaucoup de paramètres à une fonction peut se révéler lourd à l'utilisation. Le programmeur doit réfléchir à l'éventualité de créer une structure de sauvegarde pour passer les paramètres (structure, objet, etc.).

### Exemple :

```
maFonction (typeVariable1 variable1, typeVariable2 variable2,  
            typeVariable3 variable3, typeVariable4 variable4)  
|-| <- espace en trop, pas grave.
```

## 5.5 Parenthèses

Les parenthèses servent à éviter les ambiguïtés, il est donc très recommandé de les utiliser aussi souvent que nécessaire. Il n'y a pas d'espace entre une parenthèse et ce qu'elle contient.

### Exemple :

```
/* — KO — */  
Si (a == b Et c == d)  
  
/* — OK — */  
Si ((a == b) Et (c == d))
```

## 5.6 Accolades

Quelques conventions imposent l'accolade ouvrante en fin de ligne. Cependant, il est à noter que la compréhension d'un code plus aéré est plus rapide et plus simple. De plus, il est logique d'indenter les accolades au même niveau (vision de bloc). C'est pour ces raisons qu'il est conseillé d'ouvrir et de fermer les accolades sur une nouvelle ligne, en respectant l'indentation.

### Exemple :

```
/* — KO — */  
Si (condition) Alors {  
    [action]  
} sinon {  
    [action]  
}
```

```
/* — OK — */  
Si (condition) Alors  
{  
    [action]  
}  
Sinon  
{  
    [action]  
}
```

## 5.7 Espaces

Les espaces sont très importants pour la lisibilité du code car comme les éléments précédents, ils lui confèrent une homogénéité qui rendra sa lecture plus facile.

Où les espaces doivent ils apparaître ?

- après un mot clé (**for**, **while**, **switch**, **if**, etc) ;
- après une virgule (paramètre) ;
- avant et après un opérateur (+, =, <) ;
- entre une variable et son typage (cast).

# Structures de contrôle

---

Les mots clés des structures de contrôle doivent être séparés par un espace de la parenthèse ouvrante, afin de ne pas les confondre avec des appels de fonction.

Les instructions d'une structure de contrôle sont le plus souvent placées entre accolades. Cela facilite la compréhension de la structure du code.

## 6.1 Instructions

Il est très fortement conseillé de n'effectuer qu'une seule instruction par ligne.

### Exemple :

```
/* — KO — */
uneVariable++; uneAutreVariable—;

/* — OK — */
uneVariable++;
uneAutreVariable—;
```

## 6.2 Retour de fonction

Il est préférable de n'avoir qu'un seul état de retour dans une fonction. Ce qui sous-entend que l'on peut faire appel à une variable pour stocker un résultat à retourner. Cette manière de faire aide beaucoup au débogage.

### Exemple :

```
/* — KO — */
/* Commentaire */
Si (condition) Alors
{
    retourner uneVariable;
}
retourner uneAutreVariable;

/* — A éviter (opérateur ternaire cf section 6.10) — */
retourner ((condition) ? uneVariable : uneAutreVariable);

/* — OK — */
variableARetourner = ((condition) ? uneVariable : uneAutreVariable);
retourner variableARetourner; // Une seule action par ligne
```



### 6.3 Si-alors-sinon

Lorsque la portée d'un si-alors(-sinon) est relativement grande, on doit rajouter des commentaires sur les conditions pour rappeler à l'utilisateur à quel si on se réfère (notamment lorsque plusieurs si sont imbriqués). Il est bon également de commenter les actions à effectuer.

**Exemple :**

```
/* — KO — */
Si (condition1) Alors
    [action]
Sinon
    [action]

/* — OK — */
/* Commentaire sur action Si */
Si (condition1) Alors
{
    /* Commentaire sur action */
    [action]
}
/* Sinon (condition1) : Commentaire sur action sinon */
Sinon
{
    /* Commentaire sur action */
    [action]
} // Fin si (condition1)
```

### 6.4 Tant que

Un commentaire de condition peut être ajouté si cela est jugé nécessaire.

**Exemple :**

```
/* Commentaire sur l'action du Tant que */
Tant que (condition)
{
    /* Commentaire sur action */
    [action]
}
```

### 6.5 Pour

Un commentaire de condition peut être ajouté si cela est jugé nécessaire.

**Exemple :**

```
/* Commentaire sur l'action du Pour */  
Pour (initialisation ; condition ; avancement)  
{  
    /* Commentaire sur action */  
    [action]  
}
```

## 6.6 Pour chaque

Un commentaire de condition peut être ajouté si cela est jugé nécessaire.

### Exemple :

```
/* Commentaire sur l'action du Pour chaque */  
Pour chaque (tableau : numElement => element)  
{  
    /* Commentaire sur action */  
    [action]  
}
```

## 6.7 Répéter-Jusqu'à

Un commentaire de condition peut être ajouté si cela est jugé nécessaire.

### Exemple :

```
/* Commentaire sur l'action du Répéter-Jusqu'à */  
Répéter  
{  
    /* Commentaire sur action */  
    [action]  
}  
Jusqu'à (condition)
```

## 6.8 Switch

Dans un switch, il est de rigueur de ne pas oublier le résultat par défaut et les commentaires sur l'action que l'on va effectuer. De plus, si un break est omis volontairement, il doit être signalé par un commentaire.

### Exemple :

```
Switch (condition)  
{  
    case 1 :  
        /* Commentaire sur l'action */  
        [action]  
        /* Action sans break => va faire l'action suivante */
```

```
case 1 :  
    /* Commentaire sur l'action */  
    [action]  
    break;  
  
default :  
    /* Commentaire sur l'action */  
    [action]  
    break;  
}
```

## 6.9 Break & continue

L'utilisation de break et de continue est rarissime. Le plus souvent, c'est un problème de modélisation de la solution. Il faut donc se poser la question de savoir si leur utilisation est vraiment obligatoire. De plus, leur utilisation génère souvent des erreurs qui sont, par le fait, difficiles à localiser.

### Exemple :

```
/* Exemple de boucle recherchant un objet */  
  
/* KO */  
Pour (initialisation ; condition ; avancement)  
{  
    Si(objetEnCours = objetATrouver())Alors  
    {  
        objetATrouver = objetEnCours;  
        break; // Break va sortir de la boucle  
    }  
}  
  
/* OK */  
objetTouve = faux;  
Pour (initialisation ; (condition Et !objetTouve); avancement)  
{  
    Si(objetEnCours = objetATrouver())Alors  
    {  
        objetATrouver = objetEnCours;  
        objetTouve = vrai;  
    }  
}
```

## 6.10 Opérateur ternaire

Afin d'optimiser le code, on peut utiliser l'opérateur ternaire. En revanche, le code de la condition et des actions doivent être très court. La condition d'un opérateur ternaire est toujours entre parenthèses.

### Exemple :

```
/* Affectation d'une valeur à une variable */
```

`uneVariable = (condition) ? uneAutreVariable : uneAutreVariableEncore ;`

# Débogage

---

Le débogage d'une application est action récurrente chez les développeurs. Étonnament, celles-ci ne fonctionnent pas toujours très bien suite à leur développement ! Pour palier ce problème, les *echo "coucou"* peuvent résoudre plus ou moins rapidement les problèmes. Plus proprement, la mise en œuvre d'un système d'auto-audit, avec différents niveaux d'affichages selon le niveau de débogage, permettra de repérer plus rapidement les erreurs.

## 7.1 Benchmark

Le système de benchmark est un outil très puissant qui permet de repérer non seulement où l'application est défaillante mais aussi les problèmes de temps de calculs. Pour mettre en place ce genre de système, réutilisez une classe de benchmark existante et insérer dans votre code les appels à l'objet de benchmark.

### Exemple

**Fonction** typeRetourné maFonction ()

```
{  
    Benchmark("nom de la fonction", ENTREE);  
    [action]  
    Benchmark("nom de la fonction", SORTIE);  
}
```

Dans la version finale du logiciel, on effacera toutes ou parties des références au benchmark.

Voici un exemple de statistiques fournies après l'exécution d'un benchmark :

Fonction	Temps moyen d'exécution	Nb d'appels
Constructeur()	24 ms	1
Fonction1()	1 ms	2
Fonction2()	2 ms	6
SousFonction2()	3 ms	2

### 7.1.1 Réutilisation

La réutilisation de code est concept clé pour le développement efficace. En effet, l'adage dit qu'*il ne faut pas réinventer la roue*, et on ne le répètera jamais assez !

Tout développeur est amené à créer ses propres classes et méthodes en fonction d'un besoin métier bien spécifique. Mais s'il peut réutiliser un code existant, il gagnera autant de temps pour réfléchir à l'architecture de son besoin, au développement et aux tests nécessaires de ses propres composants.

N'hésitez pas à utiliser un code validé par la communauté, c'est comme ça que les connaissances partagées aident à faire avancer un système : pas de stagnation (pas de réinvention constante).

# Organisation des sources

---

La structuration des fichiers est une notion fondamentale en génie logiciel. En effet l'utilisation d'un schéma d'organisation des données offre une approche intuitive de la structuration du code et des documents associés permettant une consultation plus efficace.

Dans un projet, il est important de classer les fichiers selon leur type ou leur utilisation. Par exemple, pour un site web, il existera un dossier *images* dans lequel seront déposées toutes les images du site. Ce dossier *images* pourra également contenir des sous-dossiers afin de mieux classer les fichiers. C'est cette catégorisation hiérarchique qui aidera l'utilisateur à ordonner ses fichiers.

Il faut noter que nombre de framework utilisent une hiérarchie plus ou moins commune afin de permettre la localisation des fichiers.

## Exemple :

- css/
  - default/
  - ie/
- images/
  - animation/
  - statique/
    - png/
    - jpg/
- pages/
  - visiteur/
  - administration/
- librairies/

## Exemple (Zend Framework) :

- Library/
  - Zend/
    - Form/

# CheckStyle

---

Pour vérifier la bonne mise en œuvre des règles d'une convention, certains plugins ont été créés pour contrôler le code fait par le programmeur.

Un des plugins les plus connus est *CheckStyle*. Ce plugin fonctionne sous NetBeans et Eclipse.

## A.1 Installation

Pour installer CheckStyle :

- [Télécharger le plugin](#)
- Dans l'IDE, lui fournir le fichier de configuration

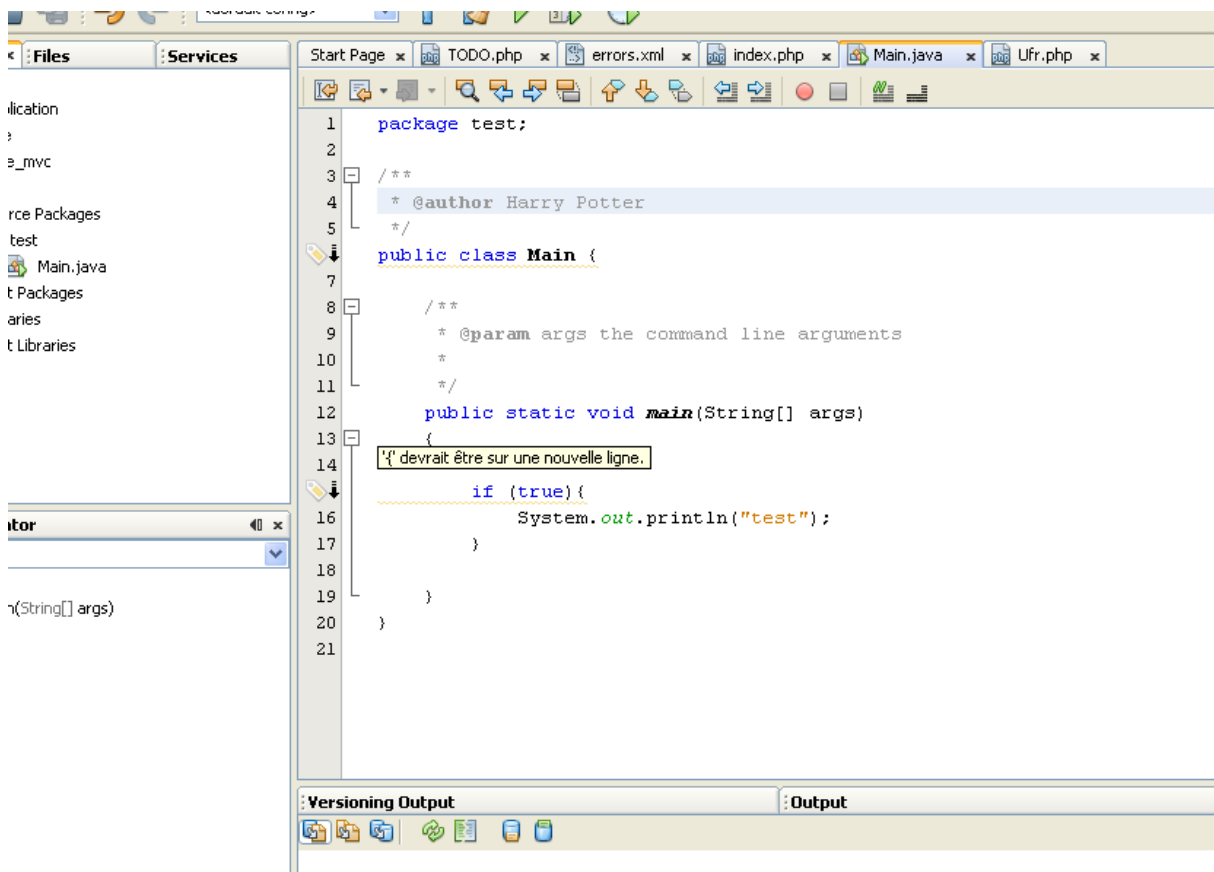


FIG. A.1 – Aperçu du plugin CheckStyle dans NetBeans

# Javadoc et Doxygen

---

## B.1 Javadoc

La Javadoc est un outil développé par Sun Microsystems [3] permettant de créer la documentation des API d'une application à partir des commentaires présents dans le code source. Au début, ce principe était exclusivement réservé au langage Java, mais très rapidement, cet outil s'est popularisé. Bon nombre d'outils permettent désormais de générer une documentation au format HTML à partir de nombreux langages.

Dictionnaire des tags de documentation Javadoc :

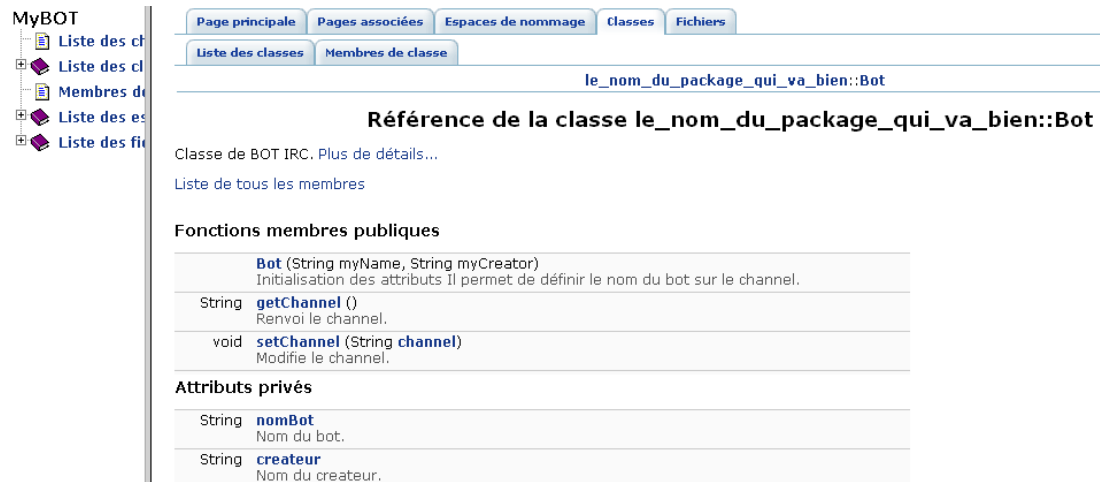
Tag Javadoc	Description
@author	Référence le développeur de la classe ou de la méthode
@version	Référence la version d'une classe ou d'une méthode
@see	Précise une référence utile à la compréhension
@param	Référence un paramètre de méthode. Requis pour chaque paramètre
@return	Référence le type de valeur de retour. A ne pas utiliser pour une méthode sans retour
@deprecated	Marque la méthode comme dépréciée (à ne pas utiliser)
@exception / @throws	Référence aux exceptions susceptibles d'être levées par la méthode

## B.2 Doxygen

Doxygen est un outil utilisé pour générer la documentation d'une application au format HTML en se basant sur les commentaires de documentation. Il tient compte de nombreux langages tels que le C, le C++, le php, le Java, etc. Pour télécharger le logiciel, vous devez vous rendre sur le site officiel [4].

Ce logiciel est très facile d'accès et intuitif. Il est rapidement devenu un standard de génération de documentation.





The screenshot shows a web-based documentation interface. On the left is a sidebar titled 'MyBOT' with a tree view containing links like 'Liste des classes', 'Liste des fichiers', 'Membres de classe', 'Liste des espaces de nommage', and 'Liste des fichiers'. The main content area has a top navigation bar with tabs: 'Page principale', 'Pages associées', 'Espaces de nommage', 'Classes', and 'Fichiers'. Below this, there are sub-tabs for 'Liste des classes' and 'Membres de classe'. The main title is 'le\_nom\_du\_package\_qui\_va\_bien::Bot'. Below the title is the heading 'Référence de la classe le\_nom\_du\_package\_qui\_va\_bien::Bot'. The text 'Classe de BOT IRC. Plus de détails...' is followed by a link 'Liste de tous les membres'. Under the heading 'Fonctions membres publiques', there are two entries: a constructor 'Bot (String myName, String myCreator)' with a description 'Initialisation des attributs Il permet de définir le nom du bot sur le channel.', and two methods: 'String getChannel ()' with description 'Renvoie le channel.' and 'void setChannel (String channel)' with description 'Modifie le channel.'. Under the heading 'Attributs privés', there are two entries: 'String nomBot' with description 'Nom du bot.' and 'String createur' with description 'Nom du createur.'.

FIG. B.1 – Aperçu d’une documentation générée par Doxygen

# Quelques illustrations

```
40 public void changeClothe(String clotheType, DrawableImage newDrawableClothe)
41 {
42     if(newDrawableClothe != null)
43     {
44         Iterator<DrawableImage> iterator = drawables.iterator();
45         while (iterator.hasNext())
46         {
47             drawableTmp = (DrawableImage) iterator.next();
48             drawableTmp.
49             if (drawable
50             {
51                 iterator
52             }
53         }
54         addDrawable(newD
55     }
56 }
57
58 public void addDrawable(
59 {
60
```




FIG. C.1 – Aperçu de l'auto-complétion sur une fonction non documentée

```
40 public void changeClothe(String clotheType, DrawableImage newDrawableClothe)
41 {
42     if(newDrawableClothe != null)
43     {
44         Iterator<DrawableImage> iterator = drawables.iterator();
45         while (iterator.hasNext())
46         {
47             drawableTmp = (DrawableImage) iterator.next();
48             drawableTmp.g
49             if (drawable
50             {
51                 iterator
52             }
53         }
54         addDrawable(newD
55     }
56 }
57
58 public void addDrawable(
59 {
60
```

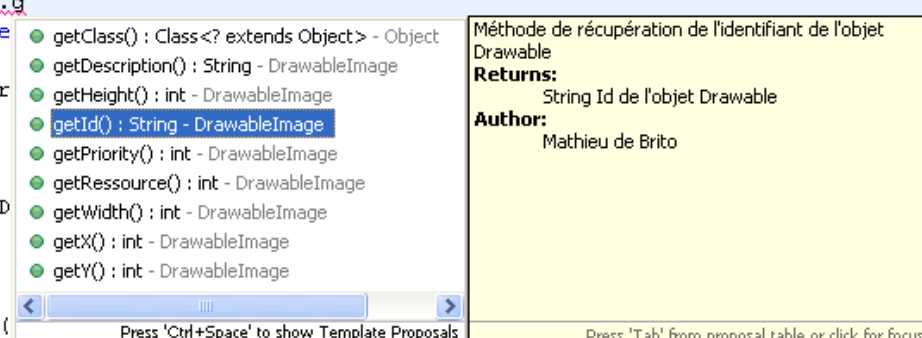


FIG. C.2 – Aperçu de l'auto-complétion sur une fonction documentée

# Bibliographie

---

- [1] Oliver Burn. Checkstyle. <http://checkstyle.sourceforge.net>.
- [2] Sun Microsystems. Convention java. <http://java.sun.com/docs/codeconv>.
- [3] Sun Microsystems. Javadoc. <http://java.sun.com/javase/downloads>.
- [4] Dimitri van Heesch. Doxygen. <http://www.doxygen.org>.
- [5] Wikipédia. Checkstyle. <http://fr.wikipedia.org/wiki/Checkstyle>.
- [6] Wikipédia. Convention de nommage. [http://fr.wikipedia.org/wiki/Convention\\_de\\_nommage](http://fr.wikipedia.org/wiki/Convention_de_nommage).
- [7] Wikipédia. Doxygen. <http://fr.wikipedia.org/wiki/Doxygen>.
- [8] Wikipédia. Génie logiciel. [fr.wikipedia.org/wiki/Génie\\_logiciel](http://fr.wikipedia.org/wiki/Génie_logiciel).
- [9] Wikipédia. Javadoc. <http://fr.wikipedia.org/wiki/Javadoc>.

# Convention de nommage

---

Département Informatique

Génie logiciel

**Résumé :** Convention de nommage pour les développements du département informatique de Polytech'Tours

**Mots clefs :** convention de nommage, programmation, génie logiciel

**Abstract:** Naming convention for developpments at the Polytech'Tours computer science department

**Keywords:** naming convention, developpment, software engineering

---

## Superviseurs

Emmanuel Néron

[emmanuel.neron@univ-tours.fr](mailto:emmanuel.neron@univ-tours.fr)

Nicolas Ragot

[nicolas.ragot@univ-tours.fr](mailto:nicolas.ragot@univ-tours.fr)

## Étudiant

Mathieu de Brito

[mathieu.debrito@etu.univ-tours.fr](mailto:mathieu.debrito@etu.univ-tours.fr)

DI5 2009 - 2010

Université François Rabelais, Tours