« LES 7 REGLES D'OR DU DEVELOPPEMENT »

Sommaire

0. INTRODUCTION

Ce document présente les 7 règles d'or du développement à appliquer dans l'ensemble des projets sur lesquels vous interviendrez.

0.1. Description du besoin

Avec l'expérience je me suis aperçu que les développements, effectués par mes confrères et moi-même, manquaient de rigueur sur de nombreux points. Cette rigueur vient souvent avec l'expérience et le travail sur soi pour fournir des livrables complets, tant dans leurs mises en forme, leurs contenus que leurs documentations. On pourrait faire une liste exhaustive de tout ce que doit fournir un développeur et de la quantité de règles incalculable qu'il doit suivre, toujours à la lettre. Cependant il n'est pas concevable qu'un développeur lise une documentation trop importante, de part le temps pris et le nombre de règles à retenir.

Pour cette raison, il est nécessaire de fixer un nombre de règles. Ni trop court, pour être suffisamment exhaustif, ni trop long, pour que le développeur n'oublie rien. Un architecte senior avec qui j'ai eu l'occasion de travailler répétait souvent, le nombre magique, c'est 7.

7 points, 7 règles, c'est ni trop court, ni trop long. J'ai donc retenu 7 règles d'or que doivent suivre les développeurs, 7 règles, ni trop court, ni trop long. Ces règles permettent d'obtenir, si elles sont suivies à la lettre, des livrables de bonnes qualités, permettant de :

- comprendre le code écrit,
- suivre la cinématique de l'exécution,
- reprendre le code,
- comprendre son évolution.

Voici l'ensemble de ces règles et les raisons du pourquoi :

- **Les Commentaires** : Le point principal, faisant toujours défaut, qui représente le socle fondamental de tout développement. C'est lui qui permet de comprendre le code, d'expliquer l'implémentation, de suivre la cinématique. Il ne doit jamais être négligé, jamais être remis à plus tard.

- **La Syntaxe** : Chacun a la sienne, toujours mieux que les autres, toujours différente. Comme la syntaxe linguistique, elle permet de suivre facilement la cinématique, et doit toujours être la même quitte à faire grogner les plus hardis. En appliquer une et une seule.

- **Les Attributs, Variables, Paramètres et Constantes** : Le nommage des objets utilisés chaque jour doit être réglé, choisi, pour permettre, comme la syntaxe, « de retrouver ses petits » comme disent certains. Pas de noms trop courts, pas de numéro, et tout se lit plus facilement.

- **La Simplicité** : Ce n'est pas parce qu'un algorithme est complexe, que le code doit l'être également. Et ce n'est pas parce que l'algorithme est simple que le code doit être complexe. Rendre les choses simples, des classes courtes, des méthodes courtes, quitte à faire plus de classes, plus de méthodes, plus d'appels, avec plus de paramètres.

- **Les Traces** : Tracer une application, afin de pouvoir la suivre plus facilement, comprendre la cinématique, et valider la bonne exécution. Ces traces, si simples à mettre en place, se doivent de respecter un minimum de règles pour être utiles.

- **Les Properties et la Globalisation** : Paramétrer son application dans un fichier, pour lui permettre de s'adapter à son environnement (un emplacement différent, une base de données différente – une petite modification dans un fichier suffira). Et le langage, un autre fichier de paramètre, parce qu'une application se doit d'être multilingue, dès le départ.

- **La Gestion de Version** : Gérer les versions, afin de faire des modifications sans crainte, afin de retrouver une modification sans difficulté, afin de suivre l'historique des développements.

0.2. Description du document

Projet :	Le développement dans son état brut
Référence :	Les 7 règles d'Or du développement v0.6.7.doc
Rédacteur(s) :	Sébastien PESME-CANSAR – iguane39@gmail.com
Responsable du contenu :	Sébastien PESME-CANSAR – iguane39@gmail.com

0.3. Historique du document

Version	Date	Rédacteur	Modifications effectuées
0.5	2008.03.20	S. PESME-CANSAR	Création
0.6	2008.09.10	S. PESME-CANSAR	Correction de coquille

0.4. Etat du document

Statut	Document pour V1.0
Etat	En lecture

0.5. Liste de diffusion

Service	Nom des destinataires	Nombre	Objet de la diffusion
WORLD	Les développeurs		**Information**

0.6. License

1. COMMENTAIRES

1.1. Langue des commentaires

Pour qu'un code soit lisible et facilement compréhensible, il doit être complété par des commentaires. Ces commentaires doivent être écrits dans la langue souhaitée par le client, par exemple le français, à défaut l'anglais. JAMAIS les commentaires ne doivent être écrits dans une autre langue (et pas de commentaires en roumain, comme on n'en verra pas non plus en japonais). Si un commentaire ne peut pas être compris par la personne qui lira le code plus tard (vraisemblablement le client ou un prestataire du client), alors ce commentaire ne sert à rien.

1.2. Commentaires utiles

On ne doit écrire que des commentaires qui vont aider le relecteur, et donc qui ont du sens et de l'intérêt. Un commentaire qui reprend uniquement le nom d'une classe, d'une méthode ou d'un champ n'est d'aucune utilité et n'aide en rien le lecteur, il peut même le perturber si le code a été changé sans changer le commentaire associé. NE PAS écrire de commentaire inutile.

De manière générale, un commentaire de classe ou de méthode doit toujours se voir accoler le nom du créateur de l'objet ainsi que la date de création (tags @created, @author et @since).

1.3. Commentaires de classe

Les commentaires de la classe doivent décrire pour quelles raisons cette classe existe :

- son utilité ;
- son fonctionnement ;
- ses fonctionnalités ;

Ce commentaire peut se réduire à quelques lignes (2 à 4 lignes) mais s'étendre jusqu'à plusieurs lignes (10 ou 20 lignes) si cette classe est le pivot de l'application.

```
/**
 * Cette classe permet de transformer les données fournies par le service d'appel XXX.
 * @since 2008.05.10
 * @author spesme
 */
```

1.4. Commentaires de méthode

De la même manière que pour la classe, les commentaires d'une méthode sont nécessaires pour décrire :

- l'utilité ;
- le fonctionnement ;

- les fonctionnalités ;

Il est également nécessaire de rajouter des informations supplémentaires afin d'aider le lecteur, principalement en décrivant :

- les paramètres en entrée, avec les valeurs possibles éventuelles (null, vide…) et les modifications effectuées par la méthode sur ces paramètres si le cas se présente ;
- les valeurs attendues en retour (en décrivant si null ou vide peuvent être retournés et dans quel cas) ;
- les exceptions éventuelles que cette méthode peut engendrer et pour quelles occasions ;

```
/**
 * Cette méthode transforme la liste d'utilisateurs au format Business en format propriétaire
 * XML de l'outil XXX.
 * @param userList Liste des utilisateurs au format Business (null possible).
 * @return la liste des utilisateurs au format XML propriétaire XXX (liste vide possible).
 * @exception UnknowUserException lorsque l'un des utilisateurs de la liste est inconnue.
 * @since 2008.05.10
 * @author spesme
 */
```

1.5. Commentaires d'attributs et constantes

Pour ces commentaires, une simple ligne expliquant l'utilité de l'objet suffira afin de ne pas surcharger le code.

1.6. Commentaires de code

A intervalle régulier, et de manière obligatoire aux parties difficilement compréhensible, il faut ajouter des commentaires, permettant au lecteur de comprendre les algorithmes mis en place. Ces commentaires ne doivent pas être un doublon du code développé, mais aider le lecteur à comprendre les boucles, les embranchements et/ou les tests et appels effectués. Mettre un commentaire décrivant uniquement le nom de la variable ne sert à rien, expliquer pour quelles raisons cette variable prend cette valeur est plus utile.
L'ensemble des commentaires d'une méthode doit permettre de comprendre l'algorithme de la méthode. Si ce n'est pas le cas, alors il manque des commentaires dans la méthode.

1.7. Commentaires TODO

Lorsque le développeur a :

- des difficultés à assimiler un algorithme ;
- un algorithme qui n'est pas complètement défini ;
- des spécifications insuffisamment détaillées ;
- aperçu des manques mis à jour lors de l'écriture du code ;
- la connaissance pertinente que sa tâche sur cette partie du développement ne peut pas être terminée ;

Il DOIT rajouter un commentaire TODO, lui permettant ainsi qu'aux autres développeurs intervenants sur ce code de retrouver rapidement et facilement, et de comprendre pour quelles raisons ce TODO a été posé.

Un TODO doit TOUJOURS être accompagné du nom de son auteur et de la date de mise en place du TODO. Il DOIT également contenir les raisons pour lesquelles le TODO a été placé ici.

Avec les outils fournis par les IDEs d'aujourd'hui, l'ensemble des tâches non terminées pourra être listé et les réponses aux questions posées lors de l'écriture du code pourront être trouvées afin de terminer ce qui doit l'être.

```
// TODO – spesme – 2008.05.10 : Le champ informant du nombre d'utilisateurs n'existe pas.
```

1.8. Codes commentés

Dans le code, on ne devra JAMAIS trouver de code commenté. Le code commenté :

- dénature ce qui a été développé ;
- détourne l'attention du lecteur ;
- perturbe lors de difficulté dans les algorithmes ;

Le correcteur se posera tout le temps la question de savoir si ce qui fonctionne est ce qui est dans le code ou ce qui est dans les commentaires. Tous les codes commentés doivent donc être effacés. Si jamais il venait le besoin de récupérer du code effacé, il pourrait de toute façon l'être grâce aux outils de gestion des révisions.

Il existe une exception pour le code commenté, si celui-ci est précédé d'un TODO permettant de revenir dessus et de pouvoir valider ou non un développement. Mais ceci doit être utilisé uniquement en cas d'absolue nécessité et le code commenté inutile doit être effacé une fois les validations effectuées.

```
// TODO – spesme – 2008.05.10 : Supprimer ce code lorsque la définition de l'algorithme sera
optimisée.
 /**
  * else {
  *     result = new Utilisateur() ;
  * }
  */
```

2. SYNTAXE

Lorsqu'on débute dans le développement ou si l'on a travaillé longtemps seul ou dans une équipe réduite sans imposer de règles de syntaxe dans ses développements, il n'est pas évident de suivre une autre syntaxe, pour de nombreuses raisons :

- on trouve sa/cette syntaxe plus lisible ;
- la syntaxe choisie n'est pas la « meilleure » ;
- le plaisir de respecter sa propre syntaxe ;

Ceux qui ne veulent pas se mettre à suivre les règles de syntaxe définies sont nombreux et souvent difficiles à convaincre.

Cependant suivre une syntaxe dans les développements informatiques, c'est un peu comme suivre la syntaxe définie dans la langue écrite tous les jours. Celui qui écrit un texte sans suivre les règles apprises à l'école commet des fautes. Il en est de même dans le développement informatique. Certes les programmes écrits peuvent fonctionner, mais pour tous les lecteurs, autre que « l'écrivain », le code est rempli d'erreurs de syntaxe qui rendent difficiles sa lecture.

Une fois une syntaxe choisie, par le chef de projet, le client ou l'entreprise d'édition, les développeurs et l'ensemble des acteurs intervenant sur le projet doivent s'y tenir et ce en toutes occasions.

JAMAIS les développeurs ne doivent choisir eux-mêmes la syntaxe qu'ils souhaitent utiliser, aux risques de rencontrer une syntaxe par développeur et de n'avoir au final qu'un code parsemé de syntaxes différentes fixées au bon vouloir de chacun.

Pour l'exemple, si une syntaxe comme celle de Sun pour Java (Sun Code Convention) est choisie, alors l'ensemble des débuts de blocs devront toujours voir leur « { » arriver en fin de ligne (pour une méthode, on obtient « myMethod() { ») et le fin de bloc défini par le caractère « } » toujours sur une nouvelle ligne. Que ce soit pour un bloc de méthode, de condition, de boucle ou de gestion des exceptions. Et ce pour TOUS les développeurs.

3. ATTRIBUTS, VARIABLES, CONSTANTES ET PARAMETRES

Le respect de certaines consignes dans la déclaration, le nommage et l'utilisation des variables et assimilés permet de :

- faciliter considérablement la lisibilité,
- la lecture,
- la compréhension des sources produites.

Suivre les quelques règles élémentaires suivantes est un mal nécessaire à respecter en toutes occasions.

3.1. Règles générales

De manière générale les attributs, variables, constantes et paramètres doivent toujours utiliser des noms supérieurs à 4 lettres, ceci afin de pouvoir :

- Comprendre l'utilité de cette variable,
- La retrouver facilement dans le code produit, sans obtenir des résultats de recherche faussés par des noms similaires d'autres variables et/ou codes,

Remarque : Cette règle peut ne pas être suivi dans certains cas compréhensibles, comme les boucles for utilisant des noms très courts tels « i », « j » ou « k ».

```
String user ;
String utilisateur ;
String usr ;
```

3.2. Attributs

Les attributs doivent être préfixés avec « m_ » (member) pour les valeurs d'instance et avec « s_ » (static) pour les valeurs statiques. Cette dénomination particulière permet de mettre en évidence rapidement les attributs afin de bien les différencier des variables et autres paramètres.

```
String m_user ; // member
String s_user ; // static
```

Attention : Un attribut est utilisé pour enregistrer une valeur de l'objet dans lequel il est incorporé. En aucun cas un attribut ne peut être utilisé pour faire transiter une valeur d'une méthode à une autre. Il doit définir une valeur réelle de l'objet.

Prenons l'exemple d'une méthode pouvant générer une erreur. Cette erreur doit être retournée par la méthode appelée, son état ne doit pas être conservé dans l'objet lui-même pour être récupéré plus tard dans une autre méthode. Ce mode de fonctionnement est trop source d'erreur, de mal-fonctionnement et mauvaise compréhension pour être utilisé. S'il y a erreur, cela doit être vu tout de suite.

3.3. Variables

Les variables devront TOUJOURS être déclarées ensemble au début du bloc qui les utilise, et JAMAIS elles ne devront être déclarées justes avant leur première utilisation.

Exemple :

- Une variable utilisée dans toute la méthode sera déclarée en début de méthode,
- Une variable utilisée dans un « if » uniquement devra être déclarée au début du bloc « if »,

```
public void myMethod(User p_user) {
   String firstname = p_user.getFirstName();
   String lastname = p_user.getLastName()
   String fullName = firstname + " " + lastname;
   if (user!=null) {
       String testingLastname = lastname.substring(2) ;
       String testingFirstname = firstname.substring(4);
       testLastname(testingLastname);
       testFirstname(testingFirstname);
   }
}
```

Pour les méthodes qui retournent une valeur, le résultat retourné sera TOUJOURS intégré dans une variable nommée « result », déclarée et initialisée en début de méthode.
Cette variable sera TOUJOURS retournée à la dernière ligne de la méthode. En effet, il est préférable d'éviter autant que possible les « return » au milieu d'une méthode. Il vaut mieux lui préférer le « result = value ; » et faire un « return result » en fin de méthode.

Cette règle rend le code beaucoup plus lisible et constant pour les multiples lecteurs que seront les développeurs, testeurs, chef d'équipe et chef de projet qui se relaieront sur l'application développée.

```
public String myMethod() {
     String result = "";
     …
     return "myValue";
     …
     result = "myValue";
     …
     return result;
}
```

3.4. Constantes

Il ne doit JAMAIS être autorisé dans le code d'utiliser des valeurs « en dur ». Cette règle doit être appliquée en toutes conditions, pour tout type de données, chaines de caractères ou valeurs numériques.

Il est TOUJOURS préférable de travailler avec des valeurs constantes, définies dans une ou plusieurs classes spécifiques.

Cette règle est également vraie lorsque l'on utilise des tableaux pouvant contenir plusieurs données à la manière d'un Bean. Les index du tableau doivent être définis en tant que constantes afin de limiter les erreurs dans la récupération et l'association des données.
Il est plus facile de comprendre ceci « myTable[USER_NAME] » que ceci « myTable[3] ».

La règle veut que les constantes soient TOUJOURS définies en majuscule, afin de bien les dissocier des autres variables. Par définition, une constante ne change JAMAIS de valeur, on doit donc lui associer les tags correspondants (« final » en java).

```
public static final String USER_TYPE_MALE = “Male”;
public static final int USER_INDEX_FIRSTNAME = 0;
…
userType = USER_TYPE_MALE;
…
myTable[USER_INDEX_FIRSTNAME] = firstname;
…
```

3.5. Paramètres

Les paramètres, utilisés dans les méthodes, doivent TOUJOURS être préfixés avec « p_ » (pour parameter). Un paramètre de méthode étant une donnée fournie en entrée, elle n’est pas sensée changer de valeurs. On essaiera de respecter cette règle autant que possible.

Dans les rares cas où cette règle ne pourra pas être respectée, on devra spécifier clairement dans les commentaires du code (la javadoc en java), que ce paramètre peut changer de valeur et dans quelles conditions.

```
/**
 * …
 * @param p_user Utilisateur à valider (p_user change de valeur si l’utilisateur est validé)
 * …
 */
public String validUser(User p_user) {
   …
}
```

4. SIMPLICITE

Lors du développement d'application, la simplicité doit être le mot d'ordre qui guide l'ensemble des participants au projet. Le contexte à beau être difficile, utiliser des algorithmes complexes, travailler sur un métier subtil, le code se doit TOUJOURS d'être simple et lisible.

Pour cela, il est nécessaire de respecter les quelques règles éditées ci-dessous :

- Il est préférable d'utiliser plusieurs classes courtes et simples plutôt qu'une seule grosse et complexe,
- Il est préférable d'utiliser plusieurs méthodes courtes et simples plutôt qu'une seule grosse et complexe,
- Une classe ne devrait JAMAIS dépasser 2000 lignes,
- Une méthode ne devrait JAMAIS dépasser 100 lignes,
- Lorsqu'un algorithme semble indépendant de la méthode dans laquelle il se trouve, alors il faut créer une autre méthode, quitte à utiliser des paramètres en entrée pour rendre générique l'utilisation de cette méthode et/ou algorithme,
- Le nom des classes et méthodes devra être simple et explicite,

5. TRACES

Rarement un développeur se retrouve en phase de production de l'application qu'il a lui-même développé. Pour de multiples raisons :

- Il est développeur, pas administrateur d'applications,
- Il a quitté le projet pour un autre projet,
- Il a changé de poste, rôle, équipe, société…

Lorsque le développeur devient responsable d'une application mise en production (et ceci arrive inévitablement un jour ou l'autre), et que cette application a un problème, il doit alors irrémédiablement aller chercher dans les traces de cette application pour vérifier son bon fonctionnement. C'est à ce moment-là qu'il pourra s'apercevoir que son code, ou le code des développeurs de l'application laisse à désirer s'il ne trouve pas dans les traces l'ensemble des réponses à toutes ses questions.

C'est pour éviter à tous les développeurs de se retrouver un jour confronté à ce genre de difficulté que les quelques règles qui suivent doivent être appliquées à la lettre.

5.1. Le formatage des traces

Et la règle la plus importante est d'avoir un format de traces, non seulement constant, mais avec le contenu impératif suivant :

- La date et l'heure sous format YYYYMMDD–HHMMSSms (voire encore plus précis si l'application le demande et le supporte),
- Le composant impacté, c'est-à-dire la classe et la méthode en question,
- Le niveau de traces impliqué,
- Un commentaire sur la trace en question,
- Un ensemble de paramètres avec leurs valeurs sous un format équivalent à parameter1=value1, parameter2=value2…

On obtient alors des traces de ce genre :

```
20080305–174538203 – myClass.myMethod() – DEBUG – Appel du WS en cours – userId=34672,
searchType=04, waitingTime=40
```

L'on voit très clairement qu'avec un niveau de traces de ce genre, différent du traditionnel « toto » ou « passage 1 », on obtient des informations claires et suffisantes pour valider le bon fonctionnement d'un appel bien déterminé.

Et obtenir un tel niveau de traces est souvent simple et rapide à mettre en place. Il suffit de créer une bête classe statique avec une méthode elle-aussi statique et de placer en paramètre de cette méthode :

- Le nom du composant,
- Le nom de la méthode,
- Le niveau de debug,
- Un commentaire en chaine de caractères,
- Et un tableau ou une liste de valeurs,

Créer un traceur prend 20 minutes maximum. Faire des appels sur ce traceur est équivalent à des appels sur des System.out.println. La solution est propre, standard et paramétrable. Aucune raison de s'en priver.

Donc, créez votre propre traceur tout simple ou intégrez un des nombreux traceurs existants du monde Open Source (tel Log4J) et ce dès le début du projet.

5.2. Le niveau de trace

L'autre inconvénient de l'utilisation des « System.out.println » comme traceur, c'est qu'ils s'impriment tout le temps, qu'on soit en développement, en test ou en production. Et on sait tous que lancer des traces, ca prend de la charge processeur, qui peut s'avérer précieuse en production. D'où l'autre avantage du traceur, même simple, gérer le niveau de trace.

Il est couramment défini 4 niveaux de traces dans les applications standards, que l'on retrouve détaillé ci-dessous :

DEBUG

Le debug sert principalement au développeur, pour valider la cinématique de ses développements et vérifier que pour chaque type de valeurs, on passe dans les bons embranchements.

INFO

Le niveau info permet de fournir des informations, importantes mais plus au niveau production qu'au niveau développement. C'est sur ce niveau qu'iront les administrateurs de l'application s'ils se retrouvent avec des erreurs en production ou des comportements inattendus.

WARNING

La warning indique un comportement anormal, mais pas une erreur. Cela sert à tracer par exemple les listes vides alors qu'elles devraient être pleines, les exceptions parfois attendues.

ERROR

L'erreur, c'est le cas qui n'aurait pas du se produire. En toute vraisemblance, les erreurs seront toujours affichées dans les traces, ceci afin de fournir aux administrateurs la vue minimum attendue sur l'application.

Il est important de mettre en place un niveau de traces conformes à ces directives. Ceci permet de minimiser le niveau de traces lorsqu'il n'a pas nécessité à être détaillé (pour des raisons de performance). A l'inverse cela permet de détailler les traces lorsqu'on a besoin d'une information précise.

On pourra même l'affiner par composant, plus tard, si besoin.

6. PROPERTIES ET GLOBALISATION

Properties et globalisation, ce sont les fichiers de paramétrage de l'application et les fichiers de ressources fournissant les contenus permettant de présenter l'application dans plusieurs langues sans avoir à modifier le code développé. Ces 2 types de fichiers sont sous format texte, facilement compréhensibles et donc modifiables par des personnes ne sachant pas développer ou ne connaissant pas l'application.

6.1. Les fichiers de properties

Plus une application est paramétrable, plus elle s'adapte à son environnement, à la machine sur laquelle elle est installée, aux clients à qui elle est dédiée, aux administrateurs qui l'ont sous leur responsabilité.

On ne peut imaginer avoir à redévelopper, retester, relivrer et redéployer une application pour laquelle il faudrait changer le lieu de sortie des traces, la connexion vers la base de données, la valeur d'une quelconque donnée insérée en dur. Ce serait un travail long, pénible, un gouffre de temps perdu en ressources humaines et un manque d'agilité grave.

C'est pour cela qu'il faut prévoir, dès le démarrage des développements, l'intégration d'un fichier de properties. Même si celui-ci ne contient que peu d'informations au départ, lui ajouter, au fur et à mesure de l'avancement dans le projet, de nouvelles propriétés afin de lui donner toujours plus d'agilité.

Il est préférable d'avoir une application avec 100 properties dans laquelle seul 20 d'entre elles sont réellement exploitées, plutôt que 5 properties paramétrables alors que 10 sont nécessaires.

De plus, une fois le fichier de properties créé et le système de récupération des paramètres implémentés, ajouter une properties supplémentaire ne prend guère que 15 minutes grand maximum. Redévelopper, restester, relivrer et redéployer, cela prend au minimum 2 heures, bien souvent 2 jours. Le calcul du temps gagné est vite fait et l'avantage indéniable.

Une application DOIT donc TOUJOURS avoir un fichier de paramétrage.

6.2. La globalisation

Globaliser son application, c'est lui donner les capacités de s'adapter à la langue de ses utilisateurs. Il n'est pas envisageable d'avoir à retoucher l'application, simplement parce qu'on veut pouvoir fournir l'application en anglais et en français.

Pour cela, il est nécessaire d'intégrer au cœur même de l'application des fichiers de ressources, contenant les textes à afficher aux utilisateurs. L'application se charge de récupérer le texte dans la bonne langue en fonction de la configuration du poste local, du choix de l'utilisateur ou d'autres règles établies dans le paramétrage.

Changer la langue de l’application devient particulièrement simple et rapide, une fois passé le difficile travail de traduction des textes.
Cette globalisation devra être considérée comme une étape obligatoire des développements afin de s’adapter aux contextes, souvent mouvants et imprévisibles de l’ensemble de nos clients.

Une application DOIT TOUJOURS avoir des fichiers de ressources.

7. GESTION DES VERSIONS

Gérer les versions, cela concerne non seulement les fichiers de développement, mais également les documents et les écritures dans des outils plus particuliers et récents mais au combien utilisés de nos jours, tels les wikis.

7.1. Les fichiers de développement

C'est la première étape du développeur. Placer toutes ses sources dans un système de gestion des révisions, afin de garder un historique des changements, centraliser et sécuriser tout le travail fourni sur le projet.

Utiliser un système comme CVS demande des règles strictes à utiliser en toutes circonstances. Voici les quelques règles à toujours valoriser :

- Insérer dans CVS uniquement des sources compilables, qui peuvent s'intégrer dans le code existant sans bloquer le travail fourni par les autres développeurs. Il n'y a rien de plus rageant pour un développeur que de faire un update sur son projet et d'obtenir une erreur de compilation qui n'a pas été vérifié par son collègue. Et c'est encore plus rageant quand le collègue en question n'est pas là pour répondre de ses erreurs.

- Fournir des commentaires lors de l'insertion de modifications dans un projet. Afin que l'ensemble des participants, chef de projet, testeurs et collègues développeurs puissent comprendre et suivre l'évolution des développements. Ceci afin de pouvoir adapter son code, prévoir les changements, anticiper sur le travail futur, bref, gagner du temps.

Une application DOIT TOUJOURS être insérée dans un système de gestion des révisions.

Un développeur DOIT TOUJOURS inséré des commentaires lors de l'intégration de modifications dans le système CVS.

7.2. La documentation

Comme bien souvent dans les projets, on travaille avec beaucoup de fichiers, de plusieurs types, écrits par différentes personnes, repris, corrigés, modifiés, adaptés… Ceci provoque les problèmes classiques :

- on ne sait plus quelle est la dernière version, qui l'a modifiée, quand, etc.,
- on se retrouve avec des formalismes de nommage différents, variables d'une personne à l'autre et variables parfois chez la même personne.

A ce moment-là, toute la documentation commence à partir en fumée. On passe son temps à chercher le dernier document, à chercher qui l'a modifié en dernier, quelles sont les modifications, etc.

On pourrait mettre en place un système simple, celui de la date de dernière modification dans le fichier, mais qui ne garantit pas encore l'unicité du document, ni même son niveau d'avancement. Si deux personnes travaillent sur le même fichier dans la même journée,

comment identifier les deux documents. Cette règle peut cependant s'appliquer pour des screenshots ou des documents remis à jour très régulièrement pour marquer un état.

On utilisera donc la numérotation simple, comme celle des versions de logiciel, sur 3 chiffres. En incrémentant IMPERATIVEMENT le numéro de version en fonction de son avancement. Ainsi même si deux personnes travaillent sur le même document le même jour, on distinguera de manière unique chaque version, et on pourra évaluer de l'avancement du document avec son numéro de version.

Cette numérotation se base sur 3 chiffres vX.Y.Z qui permettent de fournir les informations suivantes :

X : Changement majeur dans la documentation indiquant une évolution ou une étape franchie particulièrement importante et/ou impactante,

Y : Evolution importante, ayant un impact sur le contenu de la documentation,

Z : Evolution mineure, ayant très peu d'impact sur le contenu de la documentation.

Cette numérotation commencera toujours à 0.0.1 lors de l'initialisation du document et variera au cours du temps jusqu'à atteindre la version 1.0.0, première version considérée comme diffusable à l'extérieur du projet, voire utilisable en production.
Les versions X.9 indiqueront que le document passera prochainement une évolution ou une étape importante.
Il est évident que ce système de numérotation doit être utilisé par tous et en toutes occasions, que ce soit pour soi ou dans le cadre d'un projet en équipe.

```
MyProject – myDocument v0.8.14.doc
```

Il est évident également que ce système n'est pas parfait et peut provoquer encore des erreurs lors de modifications en simultané par deux personnes. Cependant il a fait ses preuves et reste de confiance pour une utilisation quotidienne soutenue au sein d'une équipe d'une dizaine de personnes.

Le système quasi-parfait sera bien évidemment le serveur de gestion de documentation (GED), centralisant et gérant la numérotation et l'incrémentation de tous les documents.

7.3. Les Wikis

Les wikis sont de formidables outils pour qui sait les utiliser. Cependant pour ne pas se retrouver avec une personne qui écrit et les autres qui lisent, il est nécessaire d'obtenir une adhésion complète à ses principes de fonctionnement et garder un minimum de rigueur.

Voici les règles à suivre :

- NE JAMAIS HESITER à écrire dans le wiki, que ce soit pour ajouter de l'information, la compléter, la corriger ou en modifier sa présentation. L'historique du wiki nous permettra toujours de revenir en arrière si le travail fourni n'est pas convaincant.
- TOUJOURS ajouter un commentaire lors de la modification des pages afin de suivre facilement les modifications faites sur ces pages,
- INFORMER ses collègues sur des modifications importantes effectuées dans le wiki,

- PREFERER l’utilisation du Wiki aux mails lors du transfert d’informations. Cela évite de devoir retrouver de l’information dispersée dans plusieurs mails envoyés par plusieurs personnes.