ANNEXE 2
Construire et conditionner un logiciel libre, guide de recommandations

Projet Serveur Libre

20 avril 2000

Résumé

Ce document décrit les recommandations de base dans l'élaboration d'un logiciel pour pouvoir le mettre en libre, c'est-à-dire pour le rendre facilement utilisable et modifiable (qualités que devrait, du reste, offrir tout logiciel, qu'il soit libre ou propriétaire). L'ensemble des conseils fournis proviennent d'une adaptation à nos besoins des recommandations GNU (GNU Coding Standards, Richard Stallman) et de l'expérience acquise lors de notre projet. Il s'en suit que ce document s'adresse plus particulièrement à des logiciels écrits en langage C dans un environnement Unix/Linux. Toutefois, on y trouvera de nombreux conseils d'ordre général qui pourront être appliqués à de nombreux langages de programmation. Ces recommandations concernent aussi bien la façon de programmer que les documentations à fournir et le contenu du fichier contenant le logiciel libre. Ces indications visent à rendre plus facile la diffusion et la reprise des logiciels mis à disposition sur le serveur de logiciel libre de l'ENST Bretagne en en proposant une présentation standard.

Table des matières

1 Introduction

Ce document est le produit de la synthèse du travail de l'équipe d'élèves de l'option IT et du mastère ISIC qui a travaillé pour permettre la mise en libre de plusieurs logiciels. Pour ce travail, nous avons repris des logiciels faits par d'autres élèves de l'école, des logiciels qui sont déjà dans le domaine du libre et les recommandations pour les logiciels du projet GNU (GNU Coding Standards, Richard Stallman).

Nous avons rencontré des problèmes de différentes natures, et nous avons estimé nécessaire de creer une liste de recommandations pour ceux qui vont écrire prochainement des logiciels avec le but de les mettre en libre.

Une partie de ces recommandations est issue directement des recommandations GNU, mais d'autres sont beaucoup plus souples, et mieux adaptées au cadre du développement logiciel à l'ENST Bretagne. Et bien sûr, nous avons ajouté des recommandations à partir de notre expérience dans le projet "Serveur Libre".

Ce document doit être pris comme une série de conseils ou des règles à suivre de façon génerale, mais chaque logiciel est un cas d'espèce, avec ses spécificités, et doit être traité comme tel.

2 Statut de ce document

Ce document n'est qu'une petite partie de ce qui peut être écrit comme recommandations pour ceux qui vont écrire des logiciels pour la communauté. Les auteurs souhaitent encourager les personnes qui le lisent à ajouter, supprimer, ou modifier le contenu de ce document.

Il est donc librement copiable, diffusable et modifiable.

3 Comportement du programme

3.1 Robustesse

Évitez les limites arbitraires sur la longueur ou le nombre de toute structure de données, y compris les noms de fichier, de lignes, de fichiers, et de symboles, en assignant toutes les structures de données dynamiquement. Par exemple, dans la plupart des utilitaires d'Unix, les lignes trop longues sont tronquées silencieusement, ce qui peut parfois créer des problèmes.

Implémenter pour chaque fonction ou appel système **la gestion d'erreur**, sauf si vous êtes sûr de vouloir ignorer ces erreurs. Incluez le texte d'erreur système (du *perror* ou de l'équivalent) dans *chaque* message d'erreur en résultat d'un appel système ayant échoué, aussi bien que le nom du fichier et le nom de l'utilitaire. Un message tel que "ne peut pas ouvrir foo.c" ou encore "mode échoué" n'est pas suffisant.

Lorsque le contrôle d'erreur détecte des conditions "impossibles" le programme doit simplement s'arrêter. Il n'y a aucune raison d'imprimer un message. Ces contrôles indiquent l'existence d'anomalies (bogues). Celui qui veut réparer les anomalies devra lire le code source et exécuter un programme de déboguage. Expliquez le problème avec des commentaires dans le code source. Les données

importantes seront dans des variables. Celles-ci sont faciles à examiner avec un debugger.

Si vous utilisez des fichiers temporaires, contrôlez la variable d'environnement TMPDIR; si cette variable est définie, utilisez le répertoire indiqué au lieu de "/tmp".

3.2 Options dans les lignes de commande

Veuillez ne pas faire dépendre le comportement d'un utilitaire du nom employé pour l'appeler. Il est utile parfois de faire un lien à un utilitaire avec un nom différent, et cela ne doit pas changer ce qu'il fait. Au lieu de cela, employez une option d'exécution ou un commutateur ou tous les deux pour choisir parmi les comportements alternatifs.

De même, ne faites pas dépendre le comportement du programme du type *d'output device* qu'il utilise. **L'indépendance vis-à-vis de l'entrée/sortie** est un principe important de la conception du système; ne la compromettez pas simplement pour éviter que quelqu'un tape une option de temps en temps (la variation de la syntaxe des messages d'erreur en fonction du terminal est par contre acceptable).

Si vous pensez qu'un *type de comportement* est plus adéquat quand la sortie est un terminal, alors qu'un autre est plus utile quand la sortie est un fichier ou un *pipe*, laissez le comportement par défaut lié au terminal, et définissez une option pour le comportement alternatif.

Pour définir un comportement alternatif d'un programme, il est intéressant de **passer des paramètres en ligne de commande**. Ces paramètres devront exister sous un format long et parlant pour rendre le programme plus convivial à utiliser (ex: '–verbose' plutôt que '-v'). Un format court alternatif pourra être défini pour les utilisateurs avertis. Pour permettre une rapide prise en main du logiciel, il est conseillé d'inclure à chaque fois un certain nombre d'options standards :

–**version :** Cette option doit amener le programme à imprimer des informations sur son nom, sa version, son origine et son statut juridique sur la sortie standard, et puis quitter directement. S'il y a d'autres options et arguments, ils doivent être ignorés, et le programme doit se terminer dès qu'il a affiché le message, sans exécuter sa fonction normale. La première ligne du programme doit être facile à analyser par le *parser*. Le numéro de version du programme doit apparaître après le dernier espace. En outre, il doit mentionner *le nom canonique* du programme, dans un format tel que : *"Gnu Emacs 19,30"*. Le nom du programme doit être **une chaîne de caractères constante**; ne le calculez pas à partir d'argv[0]. L'idée est d'énoncer le nom standard ou canonique pour le programme, et non son nom de fichier. Il y a d'autres moyens pour découvrir le nom de fichier exact à partir de la variable PATH spécifiant les chemins de recherche des commandes. Si le programme ne constitue qu'une partie d'un plus grand module, mentionnez le nom du module entre parenthèses, comme ceci : *"emacsserver (GNU Emacs) 19,30"*. Ensuite doit suivre la déclaration que le programme est un logiciel libre, et que les utilisateurs sont libres de le copier et de le modifier sous certaines conditions. Si le programme est couvert par la license GNU GPL, dites le ici. Mentionnez également qu'au-

cune garantie n'est apportée. Il est d'usage de terminer ce texte par la liste des auteurs principaux du programme. Voici un exemple de la sortie qui suit ces règles :

> *GNU Emacs 19.34.5*
> *Copyright (C) 1996 Free Software Foundation, Inc.*
> *GNU Emacs comes with NO WARRANTY,*
> *to the extent permitted by law.*
> *You may redistribute copies of GNU Emacs*
> *under the terms of the GNU General Public License.*
> *For more information about these matters,*
> *see the files named COPYING.*

–help : Cette option doit envoyer un bref descriptif sur la sortie standard expliquant la façon d'appeler le programme, puis se terminer sans erreur. D'autres options et arguments doivent alors être ignorés et le programme ne doit pas exécuter sa fonction normale. Une adresse mail pourra être fournie pour effectuer les *'bug reports'*.

Dans un souci d'uniformisation, il est conseillé de se reporter au manuel "GNU Coding Standards" pour avoir une liste exhaustive des paramètres habituels pour nommer vos autres options.

3.3 Utilisation de la mémoire

Si un programme n'utilise que quelques Mo de mémoire, ne faites pas d'effort particulier pour réduire l'utilisation de la mémoire.

Toutefois, pour des programmes comme *cat* ou *tail*, qui peuvent être utilisés avec des fichers de grande taille, il est important de ne pas fixer artificiellement de taille limite pour les fichiers traités. Par exemple, si un fichier traite les données ligne par ligne, il n'est pas utile de garder en mémoire plus d'une ligne.

4 Format des fichiers sources

La particularité des logiciels *Open Source* est que l'on donne l'autorisation de reprendre des parties de code et de modifier le programme. Or pour que cette particularité soit exploitable, il faut que les sources fournis avec le logiciel ne soient pas trop difficiles à relire. Se plonger dans les sources d'un logiciel inconnu est déjà un exercice laborieux, inutile de le compliquer avec des problèmes de mise en forme. Voici donc quelques conseils sur la façon de formater ces sources.

4.1 Commentaires des sources

Chaque programme doit commencer par un commentaire indiquant brièvement ce qu'il fait. Exemple : `'fmt - filter for simple filling of text'`.

En ce qui concerne la langue choisie pour les commentaires, voici les recommandations GNU :

> "Veuillez écrire les commentaires dans un programme de GNU en anglais, parce que l'anglais est un langage que la plupart des programmeurs dans tous les pays peuvent lire. Si vous n'écrivez pas bien

> l'anglais, écrivez les commentaires en anglais comme vous pouvez, et demander à d'autres de vous aider à les corriger. Si vous ne pouvez pas écrire des commentaires en anglais, veuillez trouver quelqu'un pour travailler avec vous et traduisez vos commentaires en anglais."

Il est bien évident que nos recommandations ne peuvent être aussi directives, étant donné que l'essentiel des travaux que nous allons intégrer au serveur seront effectuer dans le cadre de l'enseignement d'une école française. Par contre, écrire en anglais nous semble quand même une bonne chose, car cela peut faciliter, non seulement la reprise du logiciel par un plus grand nombre de personnes, mais aussi faciliter l'intégration dans un existant. Bon courage !

Veuillez mettre un commentaire avant le code de chaque fonction expliquant son action, l'action de ses arguments, et les différentes valeurs qu'ils peuvent prendre. Si un argument est utilisé de manière non standard, ou si certaines valeurs peuvent créer des erreures inattendues (ex : les chaînes de caractères contenant des interlignes), soyez sûr de bien le signaler. Expliquez également la signification de toute valeur de retour existante.

Veuillez mettre deux espaces à la fin d'une phrase dans vos commentaires, de sorte que les *commandes de phrase* d'Emacs fonctionnent correctement. Écrivez également des phrases complètes et mettez une majuscule au premier mot. Si un identificateur minuscule vient au début d'une phrase, ne le changez pas (vous parleriez alors d'une variable différente). Si vous n'aimez pas commencer une phrase avec une lettre minuscule, écrivez la phrase différemment (par exemple, " The identifier lower case is...").

Toute variable statique doit être accompagnée d'un commentaire, comme ceci :

```
/* Nonzero means truncate lines in the display;
zero means continue them. */
int truncate_lines;
```

De plus, si vous utilisez de nombreuses parenthèses et accolades, il est bon de mettre des commentaires sur la partie fermante. Ex:

```
if (test) {
.....
} // ends if (test)
```

4.2 Apparence du code source

4.2.1 recommandations générales

Veuillez déclarer explicitement tous les arguments des fonctions. Ne les omettez pas simplement parce que ce sont de simples entiers.

Les déclarations de fonction externes et les fonctions dont l'implémentation n'apparaît que plus tard dans le code doivent être regroupées en un même endroit, de préférence en début de fichier (quelque part avant la première implémentation de fonction), ou dans un fichier *d'en-tête*. Ne mettez pas de déclaration de fonction ou de variable externes à l'intérieur des fonctions.

Trop souvent on utilise la même variable locale temporaire avec un nom peu explicite (ex : temp) tout au long des différentes fonctions. Cette variable a alors

diverses utilités et change sans cesse de valeur. Il est préférable de déclarer à chaque fois une variable avec un non parlant. Ceci ne sert pas uniquement à rendre les programmes plus faciles à comprendre mais favorise aussi l'optimisation faite par les compilateurs.

Ne pas utiliser de variables ou paramètres locaux ayant le même nom que des variables globales. Pour déclarer de multiples variables d'un même type, il est plus propre de démarrer une nouvelle déclaration à chaque ligne.

péférez :

```
int foo, bar;
```

ou

```
int foo;
int bar;
```

à :

```
int foo,
    bar;
```

N'hésitez pas à utiliser des accolades et parenthèses. Ainsi à la place du code suivant :

```
if (foo)
  if (bar)
    win();
  else
    lose();
```

écrivez plutôt celui-ci :

```
if (foo)
  {
    if (bar)
      win();
    else
      lose();
  }
```

De plus, il est préférable de séparer les affectations de variables et les autres commandes. Un mauvais exemple est :

```
if ((foo = (char *) malloc (sizeof *foo)) == 0)
  fatal (“virtual memory exhausted”);
```

La version correcte serait plutôt :

```
foo = (char *) malloc (sizeof *foo);
if (foo == 0)
  fatal (“virtual memory exhausted”);
```

De nombreux exemples de ce type pourraient être montrés. Le but n'est pas ici d'en faire une liste exhaustive, mais plutôt de faire comprendre que pour permettre une bonne relecture de votre code, il faut séparer autant que possible les différentes fonctions. En effet, si le code est retouvé, deux lignes précédemment adjacentes ne le seront peut-être plus. De plus une décomposition étape par étape facilite l'implémentation. Ainsi, nous n'avons pas parlé des règles pour indenter (bien que chaque exemple l'illustre), mais cette mise en forme est indispensable.

4.2.2 nommage

Le nom des variables et des fonctions des fichiers sources font partie intégrante des commentaires. Ne choisissez pas de nom trop court (ex: i,j,p,z...), mais recherchez plutôt des noms donnant des informations pertinentes quand à leur usage. Comme dit précédemment, il semble intéressant de les écrire en anglais.....

Ainsi les abréviations ne sont pas forcément les bienvenues. Bien entendu, si des concepts reviennent tout au long d'un programme, on explique au début les quelques abréviation utilisées (ex : db = database), mais l'excès arrive vite pour une personne reprenant votre code.

Si l'on suit les recommandations précédentes, on aura souvent affaire à des variables ayant des noms très longs. Deux formats sont alors principalement employés. Je vous laisse juger le format que préfère le projet GNU en reprenant son exemple : ignore_space_change_flag != iCantReadThis. Ceci est un détail, mais si cela peut aider certains a comprendre.....

Par contre, il est préférable de ne pas créer de nom de fichier supérieur à 14 caractères. Cela évite des problèmes sur les vieux systèmes de fichiers.

5 Mise en place d'une procédure d'installation

5.1 problèmes de portabilité

5.1.1 types de système

Le mot portabilité fait référence aux différentes version d'UNIX existantes. Pour qu'un programme soit le plus utile possible, il faut le rendre le plus portable possible. Pour des systèmes qui ne sont pas *Unix-like*, tels que MS-DOS, Windows, Macintosh, VMS, ou MVS, la portabilité est souvent difficile à réaliser. Il peut donc, à part dans des cas particuliers, sembler plus rentable d'optimiser le logiciel dans sa version Unix/Linux que d'assurer une portabilité tout système.

Pour rendre la portabilité possible, il est intéressant d'utiliser les renseignements sur le système que peut fournir *Autoconf*. Cela permet de changer les différentes options de compilation. C'est pourquoi la plupart des logiciels disponibles doivent être recompilés avant utilisation.

5.1.2 types de CPU

Ce problème peut vous sembler lointain, mais il faut savoir que certaines erreurs simples de programmation peuvent entraîner une non compatibilité. Ainsi,

il est conseillé autant que possible d'utiliser les types de variables adéquats et ne pas utiliser des propriétés particulières de certains systèmes. Par exemple :

```
int c;
.....
while ((c= getchar())!=EOF )
  write (file_descriptor, &c,1);
```

Ce code, s'il est valabe pour tout processeur *little-endian* (PC essentiellement) , ne l'est pas pour pour les *big-endian*. Déclarez donc *c* en tant que *char*....

De même, evitez ceci :

```
int *p, i;
i= (int) p;
```

Ce petit bout de code peut en effet vous poser des problèmes car il joue sur la taille relatives des objets entiers et pointeurs (variable selon les processeurs).

5.1.3 Appels système

Les appels systèmes différent d'un système à un autre. Par exemple, la variable de retour de *sprintf* différe suivant les systèmes, *vfprintf* n'est pas toujours disponible, etc... Une liste plus exhaustive de ces problèmes est disponible toujours sur notre document favori "GNU Coding Standards", chapitre 5.7.

5.1.4 Internationalisation

Pour que les interfaces des programmes puissent être traduites d'une langue à une autre (*i18n* = internationalisation), il est plus pratique d'utiliser *des fonctions d'appel de texte*. Ce texte sera ensuite affiché grâce à l'utilisation d'une bibliothèque dans la langue choisie par un paramètre de configuration. En C, la fonction *gettext* permet d'assurer cette internationalisation. Sinon, chacun peut au niveau de son logiciel faire appel à des variables d'environnement regroupées dans un fichier indépendant.

5.2 Makefile et procédure d'installation

Distribuer une version de votre logiciel est plus complexe que simplement rassembler vos fichier sources dans un fichier *tar* et le mettre sur un serveur FTP. Il faut que le logiciel puisse être configuré et installé sur différents systèmes.

5.2.1 Script de configuration

Chaque logiciel doit être fourni avec un script ayant pour nom *configure*. Celui-ci décrit les types de machine et de système sur lesquels vous pouvez compiler le programme. Il mettra donc à jour les informations des *makefile* pour permettre la compilation du logiciel.

Il faut que le fichier *configure* soit édité par l'utilisateur avant de lancer la compilation. Il existe plusieurs manières de procéder.

1. Faire le lien entre un (ou plusieurs) fichier(s) de configuration avec le(s) fichier(s) de configuration du système choisi (ex : *config.h* avec *config.i386.h*

ou *config.m68k.h*). Si vous procédez ainsi, il ne faut pas laissser dans votre *tarball* (au format *tar.gz* le plus souvent) de fichier *config.h*, de façon à ce que l'utilisateur ne puisse pas compiler votre programme sans avoir executé *configure*. L'utilisateur sera donc obligé d'inspecter le(s) fichier(s) de configuration pour pouvoir compiler, ce qui peut être parfois un problème, mais qui en général aide beaucoup.

2. De même *configure* peut créer automatiquement un *Makefile* à partir d'un fichier *Makefile.in* qui contiendra les différents champs à éditer. Là encore, ne laissez pas traîner de *Makefile* pour que l'utilisateur soit obliger de lancer le script *configure.*

3. Une autre possibilité consiste à créer un fichier *config.h.in* pour que configure crée le fichier *config.h*.

Tous les fichiers générés automatiquement doivent avoir en entête un commentaire précisant la façon dont ils sont créés, pour éviter que les utilisateurs n'essayent de les changer à la main. Un fichier *config.status* pourra être écrit par le script *configure* afin de sauver les options avec lesquelles il a été executé la dernière fois, et permettre de regénérer le même *Makefile* (un *shell script* par exemple).

De nombreux points de détails pourraient ici être repris si l'on veut se conformer aux standards GNU. Le plus simple actuellement est quand même d'utiliser le programme *autoconf*, qui aborde plus ou moins automatiquement les problèmes de standardisation. Regardez la documentation de ce programme pour plus d'information.

5.2.2 Conventions pour les Makefiles

Là encore, une génération automatique est conseillée. Voici quand même quelques points importants :

1. Précisez le shell utilisé pour exécuter le script. Ex : SHELL = /bin/sh

2. Pour faciliter les modifications, comme toujours, il faut que ce fichier soit écrit de façon claire, et en particulier, il est souhaitable de ne pas faire apparaître plusieurs fois dans le makefile le même fichier. Il est préférable d'affecter des variables avec des listes de noms de fichiers, et d'utiliser ces variables.

3. Pour la gestion de dépendances, utilisez *makedepend* dans une cible spécifique.

4. Ajoutez au moins une cible *clean* pour éliminer les produits de la compilation (fichiers .o, etc). C'est aussi l'usage de prévoir des cibles *distclean* et *maintainerclean*, qui font des nettoyages plus profonds. La première doit laisser le paquetage dans l'état ou il était avant de le compiler. La deuxieme élimine aussi tous les fichiers qui peuvent être générés automatiquement (*Makefile*, *configure*) à partir d'autres fichiers. Ces fichiers ne sont pas éliminés par la cible *clean* parce qu'il y a beaucoup d'utilisateurs qui ne possèdent pas les outils pour les générer (notamment *autoconf*).

Pour la génération automatique, *automake* est l'outil conseillé par GNU. Il prend en entrée des fichiers *Makefile.am*, et genère des fichiers *Makefile*.

6 Documentation

6.1 Format du *tarball*

Le *tarball* est le fichier compressé regroupant l'ensemble des composants d'un logiciel. Il est en effet plus facile de diffuser un logiciel en un seul bloc (si la taille le permet). Ce fichier contient un seul répertoire du nom du logiciel qui contient lui-même tous les sous répertoires. Voici quelques conseils visant à standardiser l'apparence d'un logiciel afin que les utilisateurs puissent s'habituer à utiliser des logiciels libres en cours de développement.

Pour créer le *tarball*, il faut premièremernt organiser un répertoire avec la structure décrite. Il est ensuite nécessaire de s'assurer qu'aucun fichier inutile est encore présent. Ceci est particulièrement vrai pour les fichier temporaires crées lors de l'édition, les fichiers de test, les binaires, etc... (une cible du *makefile* sera particulièrement bienvenue pour faire ce travail : "*distclean*"). De plus il est à noter que certains répertoires ne doivent pas être inclus (ex : CVS).

6.1.1 Structure

Pour que la recherche d'information soit rapide, il est bon d'utiliser une structure de sous répertoire la plus standard possible. Voici la liste des répertoires les plus souvent utilisés :

src ce répertoire contient les fichiers sources du logiciel

doc rassemble la documentation

lib permet de stocker les bibliothèques spécifiques au logiciel

include contient les fichiers de déclarations (*.h) que peuvent utiliser les différents modules

bin rassemble les binaires une fois compilés.

6.1.2 Fichiers à inclure

Voici une liste de fichier qu'il peut sembler bon d'inclure à la racine du *tarball* d'un logiciel libre. Le contenu de ces fichiers permettent aux utilisateurs ou à toutes personnes souhaitant reprendre le code du logiciel de s'informer.

README ce fichier doit contenir toutes les informations utiles à l'utilisateur qui prend contact avec le logiciel. Ainsi le nom de la license, la liste des principaux auteurs, les références au site de développement, et toute autre information qui pourrait aider un nouvel utilisateur.

COPYING ce fichier est une copie de la license qui régit l'usage de votre logiciel.

INSTALL instructions d'installation. Ce fichier peut être créé en supplément du README si ce dernier est trop long afin d'accéder rapidement aux informations essentielles.

AUTHORS il faut ici simplement donner le nom des auteurs et les parties respectives sur lesquelles ils ont travaillé (les adresses *E-mail* de chacun peuvent être les bienvenues....)

CONTENTS ce fichier à deux intérêts : il permet premièrement en cas de problème d'installation de vérifier si l'un des fichiers à été malencontreusement oublié dans le *tarball*. De plus, si une personne souhaite modifier le logiciel, ce fichier apporte un gain appréciable sur le temps de compréhension de la structure du logiciel. Il suffit souvent pour ce fichier de structurer de manière logique le résultat d'un *'ls'* et d'inclure le bref descriptif présent en entête de chaque fichier.

TODO ce fichier liste tous les changements qu'il est souhaitable d'apporter au logiciel. Bien entendu, si un fichier **BUGS** est déjà inclus, on ne recopiera pas ici son contenu. Ce fichier montre plutôt quelles sont les possibilités d'extension pour l'application qui n'ont pas encore été mises en place.

BUGS Ce fichier contient une liste de tous les bugs trouvés jusqu'à présent dans une version donnée. Cela permet de montrer si un *bug report* est nécessaire lorsqu'on trouve un bug, et aussi de voir quel travail est à faire sur le logiciel ou ce qui a déjà été pris en charge par quelqu'un.

Les fichiers suivants sont des fichiers qu'il faut rajouter au *tarball* lorsque plusieurs versions du même logiciel ont été fournies. Ces fichiers doivent être remaniés, et l'entête remise à chaque changement de version. De plus, entre deux versions, il ne faut pas effacer les changements de la version précédente, mais simplement les laisser à la fin du fichier.

ChangeLog Ce fichier contient tous les changements effectués sur tous les fichiers source entre deux versions. Ce fichier est important, car il permet de repérer plus facilement les changements ayant pu introduire à leur tour des bogues non présents dans les versions précédentes. Une bonne description de la façon dont ont peut écrire ce fichier est décrite dans le document "GNU Coding Standards", chapitre 6.5.

NEWS ce fichier contient tous les changements **visibles** entre deux versions pour un utilisateur. Il contient donc certaines informtions de *ChangeLog*, mais décrites plus brièvement.

7 Conclusion

Pour finir, un conseil vraiment général : essayez de vous mettre à la place d'un futur développeur qui veut reprendre votre logiciel ; pensez à ce qu'il va faire lorsqu'il téléchargera votre logiciel, et les difficultés qu'il va rencontrer lors de cette prise de contact, et plus encore lorsqu'il arbordera la reprise.

Ce serait un bon exercice qu'avant de finir votre logiciel, vous essayiez de reprendre un logiciel de quelqu'un d'autre pour voir comment ça se passe. Bonne chance !