Un processus de développement logiciel pour l'INRIA
Version 1.2

Janet Bertot, Patricia Bournai, Emmanuel Cecchet, Maxence Guesdon,
Laurent Guillo, Nabil Layaida, François Rouaix, Olivier Zendra

6 décembre 2004

Table des matières

1 Définition du projet **6**
1.1 Objectifs . . . 6
1.1.1 Buts . . . 6
1.1.2 Outils . . . 6
1.1.3 Pratiques . . . 6
1.2 Spécifications . . . 6
1.2.1 Buts . . . 6
1.2.2 Outils . . . 7
1.2.3 Pratiques . . . 7
1.3 Conception . . . 7
1.3.1 Buts . . . 7
1.3.2 Outils . . . 8
1.3.3 Pratiques . . . 8

2 Développement **8**
2.1 Développement . . . 8
2.1.1 Buts . . . 8
2.1.2 Outils . . . 8
2.1.3 Pratiques . . . 10
2.2 Gestion de versions . . . 10
2.2.1 Buts . . . 10
2.2.2 Outils . . . 11
2.2.3 Pratiques . . . 11
2.3 Suivi de bogues . . . 11
2.3.1 Buts . . . 11
2.3.2 Outils . . . 12
2.3.3 Pratiques . . . 12
2.4 Tests . . . 12
2.4.1 Buts . . . 12
2.4.2 Outils . . . 12
2.4.3 Pratiques . . . 13
2.5 Compilation - Build . . . 14

2.5.1 Buts . . . 14
2.5.2 Outils . . . 14
2.5.3 Pratiques . . . 15
2.6 Forges . . . 15

3 Distribution **16**
3.1 Paquetage . . . 16
3.1.1 Buts . . . 16
3.1.2 Outils . . . 16
3.1.3 Pratiques . . . 16
3.2 Installation . . . 17
3.2.1 Buts . . . 17
3.2.2 Outils . . . 17
3.2.3 Pratiques . . . 17
3.3 Propriété Intellectuelle . . . 18
3.3.1 Buts . . . 18
3.3.2 Pratiques . . . 18

4 Gestion de projet **18**
4.1 Buts . . . 18
4.2 Outils . . . 19
4.3 Pratiques . . . 19

5 Support **20**
5.1 Liste de diffusion . . . 20
5.1.1 Buts . . . 20
5.1.2 Outils . . . 20
5.1.3 Pratiques . . . 20
5.2 Site Web . . . 21
5.2.1 Buts . . . 21
5.2.2 Outils . . . 21
5.2.3 Pratiques . . . 21
5.3 Documentation technique . . . 22
5.3.1 Buts . . . 22
5.3.2 Outils . . . 22
5.3.3 Pratiques . . . 22
5.4 Documentation utilisateur . . . 22
5.4.1 Buts . . . 22
5.4.2 Outils . . . 22
5.4.3 Pratiques . . . 23

Introduction

L'objectif de ce document est d'appliquer la recommandation 3 du rapport Gautrin-Martin[1], à savoir "mettre en place les éléments de base d'un processus de développement logiciel cohérent au sein de l'institut." Ainsi que préconisé dans le rapport, nous avons formé un groupe de travail national pour rédiger cette recommandation. La première étape a été de choisir une méthode de travail (voir l'annexe pour le détail). Nous avons ensuite procédé en 2 étapes : en décembre 2003 une première version de ce document a été disponible, où certaines sections étaient vides ou presque et en décembre 2004 la version complète était enfin disponible. Nous avons essayé, dans ce document, de "parler français" et donc de n'utiliser que des termes français même si leurs équivalents en anglais sont souvent beaucoup plus usités ("révision" pour "release", "suivi de bogues" pour "bug tracking" par exemple). Si certains termes ne vous paraissent pas très clairs, vous pouvez consulter la base de données disponible à l'adresse www.culture.gouv.fr :8895.

Ce document est à diffuser dans les projets et à promouvoir dans les UR par les services de support au développement logiciel. Nous encourageons bien évidemment les discussions et les retours. Chaque personne impliquée dans du développement logiciel à l'INRIA devrait lire ce document d'un oeil ouvert et critique. Nous avons besoin en effet de définir notre propre "processus de développement", car ce qui existe ailleurs n'est pas nécessairement adapté à notre culture et à nos besoins. Ce document doit continuer à vivre et peut certainement être amélioré.

Recommandations

L'activité de développement logiciel à l'INRIA est différente dans chaque projet : elle varie en fonction du domaine scientifique, de la culture propre à chaque projet, et des objectifs du développement. Il ne semble pas opportun de chercher à définir un moule auquel chacun devrait se conformer, ni pour l'instant de dégager des grandes catégories de logiciels avec des recommandations spécifiques.

Nous avons choisi une présentation matricielle pour ces recommandations (voir page 5). Les colonnes représentent les activités du développement logiciel ; nous nous abstenons de les organiser dans le temps, à la manière usuelle des traités de génie logiciel, laissant à chacun le choix de ses méthodes, mais nous suggérons néanmoins que toutes ces activités soient prises en compte lorsque l'on veut diffuser le logiciel que l'on développe. Les lignes représentent des aspects du développement, c'est-à-dire des besoins ou des objectifs. La matrice définit donc des recommandations pour des objectifs particuliers. A chaque projet d'identifier ses objectifs, et d'appliquer les recommandations correspondantes. Il appartiendra également au projet de comprendre comment ses activités s'articulent, puisque nous avons volontairement omis de recommander un "cycle" particulier.

Objectifs

Voici une description des objectifs possibles lors du travail de développement de logiciel, que l'on retrouve dans les lignes de la matrice (page 5).

Développer un prototype de recherche C'est l'activité de base du développement à l'INRIA. Une personne (chercheur, ingénieur, thésard, stagiaire) démarre un logiciel comme expérimentation, démonstration ou objet de recherche.

Travailler à plusieurs sur un logiciel La plupart des logiciels développés à l'INRIA le sont par plusieurs personnes, ce qui crée des problèmes spécifiques de travail en groupe.

[1] Cf. devel.inria.fr/pdf/Rapport-Martin-Gautrin.pdf/

Développer un logiciel diffusable Le(s) développeur(s) souhaite(nt) diffuser leur logiciel, à des collaborateurs, partenaires ou à la communauté académique ou industrielle. L'installation et l'utilisation du logiciel par d'autres personnes que leurs auteurs nécessitent des aménagements.

Diffuser du logiciel hors de l'INRIA L'activité de diffusion est plus efficace lorsqu'elle est préparée. Il y a également des contraintes d'ordre légal et administratif à respecter.

S'organiser pour répondre aux utilisateurs Si la diffusion est un succès (et aussi pour en faire un succès), il faut se préparer à l'existence d'une communauté d'utilisateurs, avec ses besoins et ses exigences.

Diffuser pour un public académique Les utilisateurs académiques ont principalement besoin de pouvoir expérimenter avec le logiciel diffusé, souvent pour l'évaluer ou le comparer.

Diffuser pour un public industriel Les utilisateurs industriels ont un environnement de travail souvent différent du nôtre (équipements informatiques différents, qualifications de personnels variées) et des exigences plus strictes pour les logiciels utilisés, en production et même en R&D (Recherche et Développement).

Diffuser pour tout publics Le public non-informaticien (non-technicien) a des exigences différentes sur la présentation des logiciels.

Cette liste pourra bien sûr évoluer dans les versions ultérieures de ce document.

Pratique et Outils

Le document présente les activités de développement pour lesquelles nous voulons définir une recommandation, regroupées par catégorie. Chaque catégorie correspond à une section ; chaque activité correspond à une sous-section, qui elle-même est composée de trois parties : le but de l'activité, les outils recommandés et les pratiques recommandées.

Le document est donc organisé selon le schéma ci-dessous :

- X. : une des catégories d'activités de développement
  - X.Y : une des activités de développement
    - X.Y.1 : Buts de cette activité
    - X.Y.2 : Outils recommandés
    - X.Y.3 : Pratiques recommandées

Nous retrouvons les activités dans les colonnes de la matrice (page 5). Les rangs de la matrice correspondent aux objectifs des développeurs pour leur logiciel.

Matrice des activités

Pour atteindre chaque objectif (ligne), nous recommandons ci-dessous des outils et pratiques pour les activités (colonnes) correspondantes. Notez que la première ligne, qui concerne le développement lui-même, est évidemment l'objectif prioritaire, et s'ajoute aux autres lignes lorsque l'on dépasse le stade du développement de prototype pour viser des objectifs plus larges.

Exemple de lecture de la matrice

Vous êtes amené à développer un logiciel avec les conditions suivantes :

- vous serez plusieurs à travailler sur ce logiciel,
- ce logiciel sera diffusé à un public uniquement académique, hors INRIA,
- vous prévoyez d'avoir beaucoup de retours de la part des utilisateurs.

En suivant la matrice, outre les 5 points de la première ligne (définir les objectifs du logiciel, développer, utiliser un outil de gestion de versions, tester et utiliser des outils pour compiler et construire une version), nous conseillons donc de :

- *Pour travailler à plusieurs*
  - utiliser une liste de diffusion pour les développeurs (ligne 2 de la matrice, section 5.1),
  - utiliser un outil de suivi de bogues (ligne 2, section 2.3),
- *Pour diffuser à un public académique hors INRIA*
  - faire un dépôt à l'APP (Agence pour la Protection des Programmes) et choisir une licence (ligne 4, section 3.3).
  - faire de la gestion de projet (ligne 3, section 4),
  - utiliser des outils pour distribuer/installer votre logiciel (ligne 3, sections 3.1, 3.2),
  - réaliser une documentation technique (ligne 3, section 5.3),
  - écrire un document de spécification et de conception (ligne 3, sections 1.2, 1.3),
- *Pour gérer beaucoup de retours de la part des utilisateurs*
  - réaliser un site Web pour promouvoir votre logiciel avec un wiki, une FAQ, etc. (ligne 4, section 5.2),
  - utiliser une liste de diffusion pour les utilisateurs (ligne 5, section 5.1),
  - utiliser un outil de suivi de bogues (ligne 5, section 2.3).

	1.1	1.2	1.3	2.1	2.2	2.3	2.4	2.5	3.1	3.2	3.3	4	5.1	5.2	5.3	5.4
Développer un prototype ou un logiciel	**X**			**X**	**X**		**X**	**X**								
Travailler à plusieurs sur un logiciel						X							X			
Développer un logiciel diffusable		X	X						X	X		X			X	
Diffuser du logiciel hors de l'INRIA											X			X		
S'organiser pour répondre aux utilisateurs						X							X			
Diffuser pour un public académique															X	
Diffuser pour un public industriel															X	X
Diffuser pour tout publics															X	X

1 Définition du projet

1.1 Objectifs

1.1.1 Buts

Définir l'objectif du logiciel, les motivations, les ressources disponibles ou envisagées, les dates marquantes.

1.1.2 Outils

Voici un exemple de squelette de document :

Utilisateurs
- A qui s'adresse le logiciel ? (académiques : quel cercle ? industriels : R&D ou production ? PME ou grosses sociétés ? France, Europe ou International ?)
- Quels sont les besoins qu'il vise à satisfaire ?
- Quel est le mode de diffusion envisagé (licences, start-up, Open Source, etc.) ?

Portée du logiciel
- Définir à très gros traits les fonctions du logiciel.
- Placer le logiciel dans un contexte plus général (logiciels compétiteurs ou complémentaires, académiques ou industriels) et définir le contour (cerner ce qui ne fera pas partie du logiciel).

Quels sont les moyens techniques envisagés ?
- Quels sont les résultats de recherche déjà connus qui seront implantés ? Quelles sont les adaptations au "monde réel" prévisibles ?
- Quels sont les résultats de recherche à intégrer dans un futur proche ?
- Quels sont les développements nécessaires non-issus directement des résultats de recherche (interface utilisateur, portages, démonstrations, documentations et tutoriels, etc.) ?

Quels sont les moyens humains envisagés ?
- Identifier les personnels (permanents ou non), leur implication (temps disponible) et leurs rôles.
- Identifier les besoins non satisfaits (p.ex. support, documentation) par les ressources naturelles du projet (permanents et thésards), et examiner les ressources possibles (personnel contractuel, etc.)

Quel est le calendrier envisagé ?
- Déterminer des dates marquantes pour la vie du logiciel, en particulier la "1.0", et si possible un plan multi-distribution (v1.0, v2.0, ...).

1.1.3 Pratiques

Il est improbable que le document d'objectifs soit écrit au début du projet dans notre contexte de recherche scientifique ; néanmoins, lorsque l'engagement de ressources devient significatif (parce que le logiciel atteint un niveau de maturité suffisant, ou lorsqu'une communauté d'utilisateurs est établie), il est important de documenter les objectifs recherchés.

1.2 Spécifications

1.2.1 Buts

Le cahier des charges décrit les besoins que doit combler le logiciel.

La phase de spécification (ou phase d'analyse) consiste à décrire ce que le logiciel doit permettre de faire, du point de vue de son utilisation, et non comment il doit le faire, cette dernière question relevant de la conception. La spécification est donc orientée *utilisateur*, *fonctionnalités*, *description de formats* et *sémantique*.

1.2.2 Outils

On peut conseiller un logiciel de traitement de texte comme LaTeX pour produire les documents de spécifications sous forme de texte en langage naturel.

On peut utiliser UML avec les cas d'utilisation (use case) et les diagrammes d'activité. Beaucoup d'outils permettent la modélisation UML (voir la sous-section 1.3.2).

1.2.3 Pratiques

Nous recommandons l'écriture d'une petite documentation indiquant ce que le logiciel doit faire et comment il fonctionne, les limites prévues, etc.

Dans le cas d'une bibliothèque, la spécification peut être simplement l'API, ou l'interface de programmation fournie par la bibliothèque, son fonctionnement interne étant du ressort de la conception.

Dans le cas d'un logiciel, on peut décrire les fonctionnalités et groupes de fonctionnalités offertes par le logiciel, dans une approche déjà modulaire qui pourra servir de base (ou pas) pour la conception. Ces fonctionnalités pourront être, selon les cas, de haut niveau ou très précises ; cela dépend notamment de la précision de l'idée que l'on a du logiciel final.

Pour chaque fonctionnalité, on décrira ses entrées et ses sorties, ainsi que, idéalement, les jeux de tests associés et utilisés lors de la phase de validation.

Les différents langages et formats de fichiers utilisés par le logiciel sont aussi définis dans la spécification, par exemple sous forme de BNF.

Quelques conseils sur les spécifications fonctionnelles :
french.joelonsoftware.com/Articles/PainlessFunctionalSpecifi-2.html.

1.3 Conception

1.3.1 Buts

Nous avons vu dans le paragraphe précédent que le rôle de l'analyse est de structurer et comprendre le système à produire pour fournir des spécifications fonctionnelles de ce système sans aucune référence à des contraintes d'implémentation. La phase de développement qui regroupe la conception et l'implémentation est conduite à partir du modèle fourni par l'analyse.

La conception permet de définir l'architecture logicielle de l'application. Une approche classique est la conception modulaire qui propose une décomposition fonctionnelle et hiérarchique du logiciel. Une seconde approche est la conception objet qui part des objets de l'application et qui décrit les propriétés de ces objets, les opérations qui leur sont associées, les intéractions qu'ils ont entre eux ou avec leur environnement, etc. Au cours de la conception, outre le découpage structurel de l'application, des décisions stratégiques d'implémentation sont prises, par exemple quel langage de programmation, quelle gestion mémoire, quelle gestion d'erreurs, quelles bibliothèques utilisées, quel environnement graphique, quelles contraintes temps-réel.

1.3.2 Outils

Pour réaliser une conception objet évolutive, il est possible de s'appuyer sur UML et ses différents diagrammes (diagrammes de séquence, d'état, de classe, ...). Il existe de nombreux environnements UML sur le marché (Rose, Objecteering, ...) et dans le monde open source (module d'extension UML pour Eclipse, Poseidon, Dia, Umbrello, ...). Ils permettent en général d'engendrer un squelette de code C++ ou Java à partir des diagrammes de classes. Les environnements payants permettent le "reverse engineering", c'est-à-dire d'engendrer des diagrammes à partir d'un code déjà existant ; tous les environnements open source ne le font pas. Dans tous les cas, la conception doit donner lieu à la rédaction d'un rapport technique de conception qui doit être aussi complet que possible.

1.3.3 Pratiques

Pendant la phase de conception, il est tout à fait possible de découvrir des points flous ou ambigus dans les spécifications du système, ce qui obligera à revenir sur l'analyse. De la même façon, pendant la phase d'implémentation, il sera peut-être nécessaire de revenir sur le modèle de conception. Rien n'est figé.

Un des changements fréquents est une modification dans l'environnement d'implémentation, ceci implique qu'un minimum d'objets doivent percevoir les contraintes dues à l'environnement. Ainsi les changements seront limités à un (ou quelques) objet(s) et ne changeront pas le comportement des autres.

2 Développement

2.1 Développement

2.1.1 Buts

L'idée est de mettre dans les mains d'un développeur des outils qui l'aident à écrire du code efficacement ; l'efficacité des algorithmes est un autre sujet qui ne sera pas traité ici. Par exemple, pour écrire du code efficacement, le développeur a besoin de naviguer parmi les classes et méthodes dans son code. Il a besoin d'outils qui lui permettent de construire son exécutable automatiquement, ou d'outils pour déboguer son code, etc. Tous ces outils vont faire en sorte que le développeur avance plus vite dans son projet de développement. De plus, il faut que chaque développeur évite de réinventer la roue. De nombreuses bibliothèques/composants logiciels sont disponibles et bien maintenus. Mieux vaut se concentrer sur la partie propre de son développement, plutôt que de récréer encore une autre bibliothèque, par exemple, pour manipuler des matrices ou pour analyser du XML.

2.1.2 Outils

Plusieurs outils peuvent être utilisés pour l'écriture du code source et de la documentation interne. L'outil peut être un simple éditeur de texte ou un environnement de développement intégré (IDE) ; ceci inclut aussi des compilateurs de langages de programmation, des débogueurs, des analyseurs de code, des extracteurs de documentation, etc.

A l'INRIA, les choix d'outils et les expériences sont très variés. De nombreux projets de recherche utilisent les "GNU tools" (emacs, gcc, gdb, gprof, etc.) comme base pour leurs développements. Il faut noter que gcc, "Gnu Compiler Collection", contient des outils et bibliothèques pour C, C++, Objective-C, Fortran et Ada. (Cf. www.gnu.org/software/gcc). Pour le choix des compilateurs de

Java, il faut rester vigilant et privilégier ceux qui génèrent du code intermédiaire (ou byte code) plutôt que du code natif.

Plusieurs projets utilisent des IDE (tels que Eclipse, Netbeans, Idea ou JBuilder) qui fournissent des environnements d'édition et de débogage et qui, en plus, fournissent des outils de visualisation des classes d'objets, etc. Les IDE offrent souvent des interfaces ou des modules d'extension pour les autres outils utilisés dans le processus de développement (par exemple, JUnit pour lancer des tests unitaires de code Java depuis l'IDE ou CVS pour coupler l'édition faite dans l'IDE au gestionnaire de versions). Eclipse, qui est un logiciel libre, semble le plus populaire dans les projets de recherche. Un tour d'horizon des différents IDE se trouve à mindprod.com/jgloss/ide.html.

Quelques projets de recherche développent leurs logiciels directement sous Windows. Les outils "Visual Studio" (Visual Basic, C++, C#, J#, Web développement, etc.) de Microsoft sont les IDE les plus adaptés et les plus répandus pour le développement sous Windows. Pour plus de détails voir lab.msdn.microsoft.com/vs2005.

Pour la documentation interne de code, les deux outils les plus répandus sont Doxygen (pour C, C++ et Java) et Javadoc (pour Java). Ce sont des *extracteurs* de documentation, qui en analysant les commentaires dans le code source, génèrent la documentation API (Application Programming Interface). Doxygen est généralement réconnu comme le meilleur extracteur de documentation, car il est capable de générer la documentation en plusieurs formats (HTML, PostScript, LaTeX, etc.). (Cf. www.doxygen.org et java.sun.com/j2se/javadoc)

Il existe plusieurs outils pour analyser des programmes. Ceux-ci sont utiles pour trouver des fuites de mémoire ou des goulots d'étranglement de calcul. Les plus utilisés sont Purify, Insure et Valgrind, et plus spécifiquement pour Java, OptimizeIt et Hyades. Pour plus de détails voir :

- Purify : www-306.ibm.com/software/awdtools/purify
- Insure : www.parasoft.com/jsp/products/home.jsp ?product=Insure
- Valgrind : valgrind.kde.org
- OptimizeIt Suite (pour Java) : www.borland.com/optimizeit
- Module d'extension d'Eclipse Hyades : www.eclipse.org/hyades

Une autre préoccupation dans l'écriture du code est la compatibilité du code sur plusieurs plateformes, c'est-à-dire comment écrire du code qui marchera sur plusieurs systèmes d'exploitation (OS). Une majorité des projets à l'institut a un héritage uniquement Unix/Linux et souhaite que les logiciels fonctionnent aussi sur les machines Windows avec un effort minimal. Une possibilité est d'utiliser "Minimalist GNU for Windows" (MinGW). MinGW est une collection, gratuite et distribuable, de "header" fichiers Windows et de bibliothèques à importer, qui, utilisée avec des outils GNU, permet la génération de programmes Windows natifs qui n'ont pas besoin des DLL externes. Pour plus d'information voir www.mingw.org/index.shtml.

D'autres solutions fournissent des environnements qui importent le monde Linux sur une machine en Windows, par exemple Cygwin et SFU (Services For Unix). Cygwin est un environnement « à la Linux » pour Windows. Il contient une DLL, qui fournit une émulation de l'API Linux, et des outils pour avoir un « look and feel » Linux. Il est très répandu. SFU fournit les interfaces, tools, shells et commandes Unix standard et inclut les services Unix standard (e.g., serveur telnet, rsh, ssh, ftp, nis). SFU repose sur un sous-système, Interix, qui tourne sur un noyau Windows dans le but de faire des « cross-platform » services pour des réseaux hétèrogenes Windows-Linux. Pour plus de détails sur ces environnements Linux pour Windows voir :

- Cygwin : www.cygwin.com
- SFU : www.microsoft.com/windows/sfu/default.asp

2.1.3 Pratiques

En plus des outils eux-mêmes, il y a des pratiques qui aident à la lisibilité, la réutilisation et la maintenance de tout développement. Plusieurs projets de recherche ont des règles de codage qui permettent de comprendre plus rapidement le code développé par quelqu'un autre. Ces règles peuvent être aussi simples que des conventions de nommage pour des classes, des variables, et des procédures dans le code. L'idée est que le débogage et/ou l'extension du code est plus rapide même si le développeur n'est pas l'auteur original du code. NB : nous incluons ici des conventions sur l'organisation de la hiérarchie des fichiers des utilisateurs (les noms de répertoires, etc.) ou des modèles de "Makefiles" qui permettent aux nouveaux arrivants d'utiliser les bibliothèques du projet dans leurs propres développements. Bien sûr, ces conventions doivent être documentés (souvent sous forme de pages web). Mais cet effort est payant dans un environnement comme celui de l'INRIA où la majorité des développeurs sont du personnel non-permanent.

En plus de ces conventions générales, nous trouvons des standards de codage. Combien d'espaces pour l'indentation dans un nouveau bloc de code ; où se placent les parenthèses et accolades dans les différentes constructions d'un langage ; comment casser des lignes très longues. Quelques exemples de conventions :

- GNU Coding Standards : www.gnu.org/prep/standards.html
- Sun's Java Code Conventions : java.sun.com/docs/codeconv

En fait, il existe des outils qui vérifient automatiquement que le code est conforme aux standards. Pour Java, le site web checkstyle.sourceforge.net fournit un tel outil. Bien sûr, il faut le configurer pour le standard suivi par le projet, mais une fois configuré il peut être utilisé comme un outil séparé ou comme module d'extension dans un IDE.

Côté conventions de codage, les formatteurs de code des IDE (comme Eclipse), sont programmables pour formatter automatiquement le code selon des conventions de codages réglables (y compris passage à la ligne, formatage des commentaires, ...)

Une autre pratique pour améliorer du code est une *revue de code*. Dans l'industrie, ce processus est très formalisé. A l'INRIA cette pratique n'est pas répandue, pourtant des revues de code peuvent éviter des problèmes plus tard. Souvent les revues formelles se font avec plusieurs interlocuteurs, chacun ayant un rôle précis, mais une revue informelle pourrait être un simple parcours du code. Quelquefois des tutorats des jeunes ingénieurs incluent des revues informelles. Ou un projet de recherche décide de faire des binômes de développeurs, où chacun regarde le code de l'autre.

Il y a des pièges à éviter lorsqu'on veut écrire du "bon" code, une présentation comique, et à l'envers, du problème se trouve dans l'article en anglais *"How to write unmaintainable code"* qui se trouve à mindprod.com/unmain.html.

2.2 Gestion de versions

2.2.1 Buts

La gestion de versions a pour but de contrôler l'évolution des sources et documents du projet. Elle consiste à garder une trace des modifications effectuées sur chaque fichier du projet, ceci afin de pouvoir notamment :

- rechercher à partir de quelle version d'un fichier un bogue apparaît, afin d'en trouver la cause,
- reconstruire le logiciel dans une version donnée.

La notion de branche de développement permet d'effectuer des modifications des mêmes fichiers en parallèle, ceci afin de faire des changements lourds sans perturber le développement "normal" du logiciel. Une branche peut également être purement et simplement abandonnée. Ceci permet

d'essayer des solutions différentes et/ou dont on n'est pas certain qu'elles vont se révéler positives, et donc pas certain de leur intégration finale.

A la notion de version s'ajoute la notion de concurrence, c'est-à-dire que plusieurs personnes peuvent modifier le même fichier en même temps. Il s'agit alors d'avoir un moyen de gérer cette situation et de fusionner les modifications, automatiquement ou semi-automatiquement.

2.2.2 Outils

CVS (Concurrent Versions System) est le plus utilisé des gestionnaires de versions actuellement. Il gère également la concurrence en fusionnant les modifications parallèles ou en indiquant les conflits éventuels à résoudre à la main (typiquement un fichier modifié au même endroit en parallèle). Il permet la manipulation de branches de développement et le marquage (*tagging*) des versions.

Il existe d'autres outils de gestion de versions, comme par exemple Subversion dont le but est de remplacer CVS en apportant les mêmes fonctionnalités plus des améliorations (renommage ou suppression de répertoires, base d'archive en locale, etc.).
Voici une liste de différents gestionnaires de versions :

- CVS : devel.inria.fr/wiki/logiciels/cvs,
- Arch : www.gnu.org/software/gnu-arch,
- Subversion : subversion.tigris.org.

Les environnements de développement de type "forge" (voir section 2.6) incluent un gestionnaire de version. Certains d'entre eux s'appuyent sur les gestionnaires mentionnés ci-dessus (e.g., GForge ou Savannah), d'autres ont développé leur propre gestionnaire de versions.
Voici une liste de différentes forges qui ont leurs *propres* gestionnaires de versions :

- Visual Source Safe (pour Windows) : msdn.microsoft.com/vstudio/previous/ssafe.
- LibreSource : libresource.inria.fr.

2.2.3 Pratiques

Un gestionnaire de versions doit être utilisé dès le début du développement d'un logiciel, même si on est seul à travailler dessus.

Une politique de marquage des versions, même simple, est recommandée. Voir par exemple : devel.inria.fr/wiki/fiches/marquage_versions.

Il est également recommandé d'utiliser la notion de branche de développement lors d'une grosse modification du logiciel, pour ne pas "casser" la version courante et en permettre des corrections mineures. Lorsque la grosse modification est au point, elle est alors intégrée au "tronc commun", c'est-à-dire à la version courante.

2.3 Suivi de bogues

2.3.1 Buts

Le but d'un outil de suivi de bogues est de garder une trace de toute anomalie rencontrée dans un logiciel, depuis sa découverte jusqu'à, si possible, sa correction. Ceci permet de contrôler l'évolution de la qualité des développements, suivant leurs différentes versions. Un utilisateur peut soumettre un bogue pour une version donnée d'un logiciel ; ce bogue est ensuite validé par une personne agréée, par exemple le responsable du logiciel, et sera assigné à une ou plusieurs personnes en charge du logiciel (développeurs). Lorsque la correction sera effectuée, le bogue pourra être fermé. En dehors des bogues, il est également possible de soumettre des améliorations du logiciel. En général, il est possible d'associer une sévérité au bogue (bloquant, majeur, normal, trivial, ...). Lorsque l'outil de

suivi de bogue est relié à un outil de gestion de tâches, il est possible d'avoir des dépendances entre bogues et tâches (besoin de fixer tel bogue pour pouvoir résoudre telle tâche).

Ces outils permettent également de sortir des rapports d'anomalies sous différents formats.

2.3.2 Outils

En général, ces outils utilisent une base de données pour stocker toutes leurs informations et les plus récents dispose d'une interface Web. Il existe de nombreux outils sur le marché, notamment gratuits, par exemple :

- Bugzilla : www.bugzilla.org
- Scarab : scarab.tigris.org
- RT : www.bestpractical.com/rt
- les traceurs de bogues incorporés dans les forges (voir section 2.6, page 15).

le plus connu étant Bugzilla.

2.3.3 Pratiques

- Systématiquement tracer tous les rapports d'anomalies ainsi que les demandes d'améliorations ; même les bogues découverts en interne par un des auteurs et corrigés immédiatement devraient être tracés dans le système.
- Lors du tri des rapports d'anomalie (erreur de l'utilisateur, mauvaise compréhension, cas d'utilisation non prévu ou indéfini, vrai bogue), il est souhaitable d'associer une priorité et un responsable pour les vrais bogues.
- Pour chaque bogue, il est souhaitable de garder l'exemple qui permet de mettre le bogue en évidence de manière reproductible ; cet exemple devrait être conservé dans les tests de non-régression (voir section 2.4.3, page 13).
- Avant une révision (par exemple lorsque l'on décide d'une date pour la prochaine révision), il est souhaitable d'examiner la liste de tous les bogues ouverts et de décider ceux qui doivent être absolument corrigés ("show stoppers"), et d'indiquer pour les autres un horizon de correction, par exemple en associant la résolution d'un bogue à des tâches qui lui sont attachées.

Nous recommandons Bugzilla qui apparait le plus attractif actuellement. Un serveur national INRIA est disponible pour tous les projets de développement à l'adresse bugzilla.inria.fr

2.4 Tests

2.4.1 Buts

S'assurer de la qualité du code.

Nous pouvons considérer deux aspects :

- **Tests fonctionnels** : les fonctionnalités sont présentes et correctes,
- **Tests de non-régression** : les extensions ou les corrections de bogues n'introduisent pas de nouveaux bogues dans du code qui marchait bien jusque-là (non-régression).

2.4.2 Outils

Les (très nombreux) outils utiles pour les tests peuvent se diviser en deux catégories.

La première ("correction") consiste en des outils qui servent à vérifier une fonctionnalité du programme, une unité (classe, module, ...), ce qui se fait souvent en générant un programme effectuant **un test** bien précis. Il s'agit des tests proprement dits.

La seconde catégorie ("complétude") comprend les outils qui servent à **mesurer la couverture** des tests, c'est-à-dire quelle partie du code est effectivement testée par les moyens du paragraphe précédent. Il s'agit donc plutôt d'outils de gestion des tests.

L'importance des tests de programmes est attestée par l'énorme quantité de produits disponibles appartenant à l'une ou l'autre de ces catégories, voire les deux. L'offre commerciale dans ce domaine est considérable, mais l'offre gratuite et/ou logiciel libre est également conséquente. Il est donc difficile de conseiller ici un outil plutôt qu'un autre, d'autant plus qu'ils sont assez souvent liés à un langage particulier. Voici quelques points de départ pour rechercher un outil adapté à un langage et un besoin particulier :

- www.swtech.com/java/testing (tests et couverture pour Java),
- www.laatuk.com/tools/testing_tools.html (pour divers systèmes et langages),
- www.opensourcetesting.org (idem),
- www.qalinks.com/Tools (idem),
- www.iturls.com/English/SoftwareEngineering/SE_d5.asp (idem).

Bien entendu, l'expérience de développeurs codant dans le même langage (voire le même domaine) que vous est souvent une source d'information très efficace pour trouver un outil adapté. Il est recommandé de fréquenter les groupes de discussion `comp.lang.mon_langage`.

Avant tout ces produits, on trouve aussi, au moins pour les tests unitaires, des outils plus primitifs mais déjà installés : les langages de scripts.

2.4.3 Pratiques

Les aspects liés aux tests de programme sont très souvent négligés, car considérés à tort comme un peu annexes par rapport à l'implantation proprement dite. Il n'en est rien ! Lorsque l'on développe un logiciel qui a des utilisateurs potentiellement nombreux et/ou non experts, que l'on doit le maintenir et le faire évoluer pendant des années, avoir des **tests automatisés** et aussi systématiques que possible fait gagner un temps considérable, en permettant de repérer (et donc de corriger) les bogues très rapidement, dès qu'ils sont introduits.

Il est donc recommandé, pour chaque ajout de fonctionnalité ou d'API, d'**écrire immédiatement le ou les petits programmes de tests unitaires permettant de vérifier que la fonctionnalité marche bien** comme prévu. **L'ensemble de ces tests** unitaires (pour toutes les fontionnalités) **doivent être lancés aussi souvent que possible**, idéalement après chaque évolution du logiciel, ou chaque nuit si l'exécution de tous les tests prend trop de temps. Ceci permettra de vérifier que la qualité globale du logiciel s'améliore et que l'on n'a pas cassé une fonctionnalité jusque-là correcte en en introduisant une nouvelle (non-régression).

Notons que l'écriture d'exemples démonstratifs sert à la fois le but pédagogique (tutorial) et le test de fonctionnalités (pour autant que les exemples soient intégrés à la suite de tests).

De même, **lorsque qu'un bogue est signalé, il est indispensable d'écrire un test** aussi petit que possible qui déclenche le problème, **et de l'intégrer à la suite de tests exécutés automatiquement**[2]. Ceci évite que le bogue soit corrigé dans une version mais réapparaisse dans une version suivante sans que l'on s'en aperçoive (non-régression). Cela peut paraître inutile, mais ce cas de figure se produit en pratique bien plus souvent que l'on ne l'imagine, surtout sur un gros développement, maintenu dans la durée et par diverses personnes.

Il est souhaitable d'utiliser des environnements de développement ou des outils intégrés permettant une gestion des tests plus facile, voire partiellement automatisée (cf. les URL données en 2.4.2).

[2] Voir aussi la section 2.3 page 11.

Ils peuvent aussi offrir des fonctionalités plus avancées, comme d'indiquer le code couvert par les tests, qui sont appréciables, surtout lorsque le code devient gros.

Si de tels outils ne peuvent être utilisés, on peut éventuellement se rabattre sur des shell-scripts, avec tous leurs inconvénients (oublis, bogues, ...). Dans ce cas, il est judicieux d'écrire des tests silencieux (ou presque) lorsque tout se passe bien et qui émettent un message d'erreur sinon. Tous les tests sont automatiquement lancés par un script qui compare (avec un `diff`) la sortie obtenue à la sortie idéale et signale les différences. Bien entendu, la sortie idéale (sous forme textuelle) doit être mise à jour à chaque ajout d'un nouveau test unitaire.

Les tests et les éventuels scripts de gestion doivent être considérés comme partie intégrante du code développé. Ils doivent donc également être sauvegardés, accessibles par tous les développeurs et versionnés (cf. section 2.2 page 10).

2.5 Compilation - Build

2.5.1 Buts

Etre en mesure de compiler l'application, de construire une version exécutable et parfois, produire une distribution de manière systématique et rapide. Le build de compilation sert à décrire la chaîne de traitements qui, partant du code source, produit l'image binaire ou exécutable de l'application. Cette description comprend une déclaration explicite des dépendances entre ces traitements de façon à ce que, à chaque modification d'un module, le sous-ensemble des traitements affectés par ces changements est exécuté à nouveau (build incrémental).

Le build est souvent utilisé à d'autres fins que la compilation proprement dite. Souvent ce dernier est accompagné d'un langage de scripts (sh pour make et java pour ant), il permet donc de lancer l'exécution de n'importe quelle tâche qui intervient dans le projet. Par exemple, les tests, la génération de la documentation, la construction d'une distribution binaire ou encore l'installation sur la machine cible sont des exemples de tâches qui sont souvent écrites sous forme de scripts.

Bien que la compilation demeure le premier objectif du build, il existe des fonctions en amont du build qui aident à rendre ce dernier plus efficace. Ces fonctions ont un impact direct sur la portabilité de l'application que ce soit entre développeurs ou entre plates-formes d'exécution. De façon non exhaustive, nous énumérons les fonctions suivantes :

- analyse et gestion automatique des dépendances du code (par exemple, les dépendances entre fichiers .h et .c),
- analyse automatique et adaptation des traitements au type et à la configuration du système cible (versions du système, localisation des outils de compilation, etc.),
- vérification automatique des dépendances de l'application avec les bibliothèques (versions de bibliothèques, localisation des bibliothèques, etc.).

2.5.2 Outils

Ces outils sont :

- make : www.gnu.org/software/make/make.html
  L'outil de build le plus utilisé dans le monde UNIX. Il est également disponible sous Windows à travers les environnements Cygwin, www.cygwin.org, et MinGW, www.mingw.org.
- ant : ant.apache.org
  L'outil le plus utilisé pour le langage Java. Il est souvent intégré dans des IDE pour Java.
- autotools : les outils générateurs de build de GNU
  Il s'agit en réalité d'une famille d'outils : automake, autoconf, libtool, etc. Ces outils existent

aussi sous MinGW et Cygwin pour les plates-formes Windows.
- SCons : www.scons.org
  Un outil de construction automatique, écrit en Python, et qui utilise Python pour sa configuration. SCons se veut un outil multi-plateforme alternatif à make : il intègre en particulier des fonctionnalités similaires à autotools. Une version Windows de l'outil est disponible.

2.5.3 Pratiques

Dès l'écriture du premier fichier source, il faut identifier les niveaux de fonctions de build souhaités pour la suite du projet et concevoir le build ou le générateur de build correspondant. La pratique la plus répandue à l'INRIA est l'utilisation de make accompagné d'un certain nombre de scripts. Pour être efficace, cette démarche nécessite un paramétrage du Makefile aux différentes plates-formes. Ce paramétrage est souvent réalisé sous formes de fichiers qui sont inclus dans le Makefile principal (primitive `include`, par exemple `include Makefile.linux`).

En ce qui concerne le langage Java, l'outil le plus utilisé est ant. Il fournit également un support pour des scripts écrits en Java.

Pour les petits projets, un Makefile crée de façon manuelle peut paraître suffisant. En particulier, ceux qui ont peu de dépendances avec des bibliothèques externes. Il est néanmoins préférable de générer ces Makefiles avec les outils mentionnés plus haut. En effet, plus le projet est simple et peu dépendant de l'environnement, plus la génération automatique du Makefile est facile. De plus, il faut se rappeler que les générateurs fournissent de façon automatique les tâches de builds les plus usuelles, une raison supplémentaire pour les utiliser.

Les outils de génération de Makefiles sont relativement complexes et l'étendue de leurs fonctions est très large. Il est d'usage de commencer un projet au moyen d'un fichier automake Makefile.am et de l'adapter de façon progressive à ses besoins. A partir de ce seul fichier, il est possible de générer avec la commande automake le script configure et le Makefile contenant les tâches les plus usuelles de make comme : all, clean, dist et tags. Pour plus de paramétrage du processus de génération du build, des fichiers de configuration additionnels peuvent être utilisés : Makefile.in, config.h.in et configure.in.

2.6 Forges

Les forges sont des environnements de développement collaboratifs offrant des fonctionnalités pour le développement logiciel : gestion de configuration, de bogues, de listes de diffusion, de requêtes utilisateurs, de tâches à accomplir, outils de communication, wiki, etc. Toute la gestion de projet de développement est fait à travers une interface web. Ces environnements offrent un ensemble d'outils intégrés favorisant la collaboration.

La première forge était `sourceforge.net`, dont le logiciel sous-jacent est devenu payant par la suite. De nouvelles versions dérivées de la dernière version libre de Sourceforge ont vu le jour, principalement deux : Savannah et GForge, qui est le plus répandu. Un nouveau logiciel LibreSource a été développé à l'INRIA et entre actuellement en phase d'expérimentation.
Pour plus d'informations :
- Savannah : savannah.gnu.org.
- GForge : www.gforge.org.
- LibreSource : libresource.inria.fr.
- Visual Source Safe (pour Windows) : msdn.microsoft.com/vstudio/previous/ssafe.

3 Distribution

3.1 Paquetage

3.1.1 Buts

Être capable de distribuer le logiciel pour que le destinataire soit en mesure de l'utiliser.

3.1.2 Outils

Les outils utilisés dépendent fortement de la culture des développeurs et des utilisateurs, et de la plate-forme visée.

Les outils d'archivage sont à la fois les plus simples à utiliser et les plus universels. Le format tar, compressé avec gzip (tar.gz, le plus courant) ou bzip2 (tar.bz2, le plus efficace), permet de créer des archives destinées au monde Unix. Le format zip, qui peut éventuellement être rendu auto-extractible pour les utilisateurs ne disposant pas de l'outil de décompression approprié, est plus courant dans le monde Windows. Il est peu adapté au monde Unix, dans la mesure où il n'assure par la conservation des permissions des fichiers.

Ensuite viennent les outils de génération d'installateur graphique, comme InstallShield, InstallAnywhere ou IzPack. Le premier est un outil commercial largement utilisé dans le monde Windows. Les deux autres sont en Java, donc multi-plateforme, commercial pour l'un et libre pour l'autre. Ce dernier s'intègre d'ailleurs très bien dans un processus de compilation classique dirigé par ant ou make. Ils conviennent parfaitement à un utilisateur final, qui pourra interactivement choisir d'installer ou non telle ou telle partie du logiciel. Ils rendent par contre difficile le travail des tiers travaillant dans des environnements non interactifs.

Enfin, il faut déconseiller les outils utilisés par les distributions Linux (rpm, dpkg, ebuild) ou BSD (ports). En effet, l'intérêt des paquetages créés à l'aide de ces outils tient principalement au travail d'intégration avec le reste de leur environnement de destination, qui correspond en fait à une hyper-spécialisation de leur usage. De plus, ces distributions possèdent des moyens de diffusion (miroirs ftp publics officiels) hors de portée des développeurs. Il n'est donc guère intéressant pour ceux-ci de vouloir s'occuper eux-mêmes de tels paquetages, mais beaucoup plus avantageux de déléguer cette tâche à des tiers, tels que les contributeurs de ces distributions, en s'attachant par contre à leur faciliter la tâche. Ceci passe par l'utilisation de pratiques saines telles que celles décrites dans le wiki[3].

3.1.3 Pratiques

A l'image des outils, les pratiques dépendent fortement de la culture. Si ceci est justifié pour certains aspects, comme le choix du type de distribution, cela l'est largement moins pour d'autres, comme le choix du contenu de celle-ci.

Type Un logiciel peut se distribuer sous deux formes, source ou binaire. Une distribution source est en général indépendante de toute plate-forme, alors qu'une distribution binaire ne sera compilée que pour une plate-forme, voire une architecture de processeur, précise. Ce qui n'est pas sans poser un certain nombre de problèmes.

Dans le cas des plates-formes Windows et MacOs, l'homogénéité de ces cibles rend possible de proposer une seule distribution binaire, et le profil typique d'utilisation rendrait au contraire très difficile la compilation d'une distribution source. La distribution binaire est donc préférable.

[3] `http ://devel.inria.fr/wiki`

Dans le cas du monde Unix, au contraire, l'énorme hétérogénéité de la cible oblige à recourir à des compromis techniques peu élégants, et rend impossible de couvrir l'intégralité du spectre. Par contre, la présence standard de compilateurs, et l'existence d'outils dédiés à la portabilité, comme autotools par exemple, fait qu'il est relativement simple de proposer une distribution source universelle. La distribution source est donc préférable, dans la mesure où elle est compatible avec les modalités juridiques de diffusion.

Contenu Le contenu d'une distribution dépend fortement de son type. Une distribution source doit comprendre tout ce qui est nécessaire à la construction du logiciel et de sa documentation : intégralité des sources, scripts de compilations (configure, build.xml, makefile), etc., à l'exception des logiciels tiers. Ces dépendances doivent être mentionnées explicitement, avec la version précise dans les instructions d'installation plutôt qu'incluses dans la distribution source. Une distribution binaire, au contraire, doit être prête à l'emploi. Il est donc plus acceptable d'inclure les dépendances externes nécessaire à son fonctionnement.

En revanche, quel que soit son type, toute distribution doit inclure un fichier contenant la licence du logiciel ainsi que celles des logiciels tiers inclus si nécessaire.

3.2 Installation

3.2.1 Buts

Etre capable d'installer la version diffusée du logiciel sur une machine.

3.2.2 Outils

Les paquetages utilisant des outils avancés (e.g., rpm, dbm, InstallShield, IzPack) incluent le processus d'installation dans le paquetage. Ce processus d'installation est en général fait de scripts. Les archives classiques (e.g., tar, zip) incluent des scripts d'installation ou une cible de compilation appelée **install** (e.g., make install, ant install).

3.2.3 Pratiques

Nous recommandons l'utilisation d'outils avancés comme rpm, IzPack ou InstallShield car ils permettent non seulement l'installation plus ou moins personnalisée du logiciel sur une machine mais surtout ils permettent sa désinstallation de façon aisée en laissant la machine dans un état stable.

Il est important d'utiliser uniquement des commandes standard qui pourront être exécutées sur toute machine de l'environnement cible visé. L'utilisation d'outils comme Configure facilite la détection des outils disponibles pour générer un programme d'installation approprié.

Les répertoires d'installation doivent être configurables et le processus d'installation ne doit pas, si possible, nécessiter des droits privilégiés (root ou Administrateur). Les scripts de démarrage/arrêt du logiciel pourront se baser sur des variables d'environnement et être générés par le processus d'installation.

Le format d'encodage des fichiers entre Unix et Windows doit être préservé pour éviter tout dysfonctionnement. Les commandes unix2dos et dos2unix permettent de convertir les fichiers d'un format à l'autre. Les droits des fichiers Unix, notamment les droits d'exécution, propriétaire et groupe auxquels appartiennent les fichiers devront être choisis soigneusement.

La mise à jour d'un logiciel peut être complexe en fonction des besoins. La désinstallation des versions précédentes est souvent le plus simple (surtout en cas d'incompatibilité), mais il est souvent

nécessaire de préserver les fichiers de configuration et de les convertir à de nouveaux formats. Dans tous les cas, il est nécessaire de faire une copie de l'ancienne configuration pour être capable de revenir en arrière en cas de problème.

3.3 Propriété Intellectuelle

3.3.1 Buts

La propriété intellectuelle permet de garantir le respect des droits d'auteur et les possibilités de transfert. Une façon de protéger juridiquement son logiciel est de le déposer à l'APP (Agence pour la Protection des Programmes). L'APP est une organisation européenne privée dont l'objet est de faciliter la protection des logiciels. Un dépôt à l'APP permet de démontrer l'existence du logiciel, de lui conférer une date et de faciliter la sanction pour contrefaçon. Pour plus de renseignements, vous pouvez consulter le site de l'APP, app.legalis.net. Le dépôt à l'APP ne crée pas de droit supplémentaire vis-à-vis du logiciel. Cependant, cette formalité est toujours utile, que l'on décide ou non de diffuser le logiciel, car cela prouve la "paternité" de l'INRIA.

Une licence est *absolument* nécessaire pour *toute* diffusion d'un logiciel. Une licence est donc toujours nécessaire même pour une diffusion pour évaluation auprès de la communauté scientifique. Les deux types principaux sont les licences "libres" et les licences "propriétaire". Plusieurs types de licence libre existent ; actuellement, il est préférable de choisir une licence canonique plutôt qu'une licence spécifique. Dans le choix de la licence il faut considérer à la fois l'aspect *héréditaire* au caractère libre du logiciel et l'aspect *contaminant*. Le choix de la licence peut être contraint par l'utilisation de bibliothèques ayant leur propre licence. Par exemple, la licence open source GPL (GNU Public Licence) est contaminant et tout logiciel incluant une librairie GPL doit lui-même être GPL.

3.3.2 Pratiques

L'INRIA recommande le dépôt APP systématique et le choix d'une licence avant diffusion d'un logiciel. Sur site intranet de la DirDRI, www.inria.fr/interne/dirdri, vous trouverez les procédures à suivre et les éléments sur le choix de licence.

Etant donnée le pléthore des licences libres et la complexité du logiciel lui-même, qu'il soit dérivé des logiciels existants, ou qu'il appuie sur des bibliothèques existantes voire externes, le choix d'une licence doit être fait/validé avec le correspondant développement et rélations industrielles (CDRI) de l'UR. Le site interne de la DirDRI contient des informations utiles qui vous permettrez de voir clair dans le choix d'une licence libre.

L'utilisation ou la contribution à des projets existants, notamment open source, peut avoir des obligations de recontributions ou des implications en terme de licence ou de diffusion. D'une manière générale, les implications d'une diffusion de logiciel en matière de propriété intellectuelle sont complexes. Nous recommandons de lire la section *Logiciel* sur l'intranet de la DirDRI.

4 Gestion de projet

4.1 Buts

Dans le cadre du développement logiciel, le terme "gestion de projet" regroupe l'ensemble des méthodes qui permettent de respecter les objectifs de délais, de coûts et de qualité. La gestion de projet repose sur :

- une formalisation du déroulement du projet, sous forme de jalons et de phases,
- un suivi régulier de l'avancement des activités,
- une boucle de réaction rapide qui permet d'intégrer les aléas du projet.

Les phases d'un projet sont composées des différentes étapes explicitées dans ce document (spécification, conception, réalisation, tests), chaque phase pouvant être découpée en sous-tâches, éventuellement parallèles. Un jalon marque une fin d'étape majeure dans le déroulement du projet. Par exemple, cela peut être le début de la réalisation, une intégration avec un partenaire, ou les versions intermédiaires (alpha, beta, ...). Il se matérialise par une tenue de revue du projet qui permet de :

- constater les résultats obtenus,
- analyser leur conformité par rapport aux objectifs définis,
- décider si nécessaire des modifications à prendre pour minimiser les risques de dérive (changement d'objectifs, changements techniques, ...),
- passer en revue les critères d'acception du jalon.

Le jalon peut être accepté même si tous les critères ne sont pas satisfaits ; dans ce cas-là, il faudra préciser si ces derniers sont rajoutés au prochain jalon ou supprimés.

Un site intéressant et assez complet est celui du CNRS sur la conduite de projet : www.dsi.cnrs.fr/conduite-projet/

4.2 Outils

En général, les outils de gestion de tâches permettent de créer des projets, de les décomposer en sous-projets et sous-tâches, de marquer les jalons, de déterminer des dates de début et de fin, de mettre des dépendances entre les tâches, d'affecter une personne à une ou plusieurs tâches, de modifier les tâches tout en disposant d'un historique, etc. Tout ceci permet d'avoir une vision précise du travail en cours.

L'outil payant le plus connu est MS Project, www.msproject.com, mais il en existe beaucoup d'autres. Il en existe aussi dans le domaine open source, par exemple dotProject, www.dotproject.net. Certains outils sont incorporés dans des environnements de développement collaboratif tel que GForge, gforge.org, ou des IDE comme Eclipse, eclipse.org.

Il peut être intéressant de tirer parti d'outils collaboratifs (e.g., les wikis) pour rassembler les documents tels que comptes-rendus de réunions, de revue de projet, etc. La recherche d'informations et les mises à jour sont ainsi facilitées.

4.3 Pratiques

Il est possible de faire de la bonne gestion de projet avec ou sans outil, mais dans tous les cas, une gestion de projet est nécessaire pour éviter les dérives dans les délais ou dans les objectifs. Il s'agit d'abord d'avoir une liste détaillée des tâches, avec une estimation du temps et du nombre de personnes nécessaires pour réaliser ces tâches en se fixant un calendrier à plusieurs mois. A l'approche d'un jalon, les réunions d'avancement peuvent être plus fréquentes. Il est conseillé de rédiger des comptes-rendus de réunions. Leur ensemble forme un journal de projet où sont rassemblés tous les évènements concernant le projet (gestion des ressources, "to-do list", etc). L'outil ne remplacera ni les réunions qui permettent de faire le point, ni les documents nécessaires à chaque étape du développement mais permet d'avoir une bonne vision globale du projet ainsi qu'une vision précise des tâches en cours.

Il est nécessaire d'afficher la liste des tâches et l'échéancier surtout pour les développements comprenants des équipes délocalisées. Dans le cadre de projet open source, afficher l'ensemble des tâches (y compris celles non affectées) permet aux utilisateurs de comprendre le plan d'évolution du

logiciel et éventuellement de contribuer à des tâches non encore attribuées. Il est d'ailleurs commun dans la communauté open source d'impliquer les utilisateurs dans la définition des tâches et des jalons.

5 Support

5.1 Liste de diffusion

5.1.1 Buts

Il s'agit de faciliter la communication (écrite) :
- entre les développeurs du logiciel,
- entre les utilisateurs du logiciel et les développeurs,
- entre les utilisateurs.

5.1.2 Outils

Le service de listes de diffusion Sympa est un bon outil pour gérer des listes de diffusion. Il permet, par l'intermédiaire d'une interface web, de créer et de gérer ses listes de diffusion : droits d'accès, en-tête, etc. Cependant il ne permet pas d'indexer les archives et de les visualiser d'une façon convenable, les "mail archivers" sont des outils qui permettent d'indexer les archives et de générer des pages web. Hypermail et MHonArc sont deux exemples ; MHonArc est le plus répandu. Pour plus d'informations :
- Sympa : devel.inria.fr/wiki/fiches/sympa
- Hypermail : www.hypermail.org
- MHonArc : www.mhonarc.org

5.1.3 Pratiques

Il est préférable de n'avoir au départ qu'une seule liste de diffusion, qui sera scindée en plusieurs quand le traffic sera conséquent. Voici un exemple, pour le logiciel "foobar" :
- **foobar-dev** La liste de diffusion destinée à la communication entre les développeurs du logiciel. Les outils de gestion de tâche peuvent envoyer des mails automatiquement à cette liste.
- **foobar-users** La liste de diffusion consacrée aux questions et suggestions des utilisateurs. Cette liste doit être publique, notamment pour que des utilisateurs puissent répondre à d'autres utilisateurs, ceci afin de soulager en partie les auteurs du logiciel de cette charge. Les discussions sur les contributions proposées, les améliorations demandées ou les bogues rencontrés devraient avoir lieu sur cette liste afin de donner envie à des contributeurs potentiels de participer au développemement. Les outils de suivi des bogues peuvent envoyer des messages automatiquement à cette liste.
- **foobar-cvs** La liste de diffusion sur laquelle les messages de "commit" dans les sources du logiciel sont envoyés. Il doit être possible à n'importe qui de s'y inscrire, mais seuls les développeurs doivent pouvoir y poster des messages.

Enfin, il est recommandé d'activer la fonction d'archivage des messages, ceci afin de permettre à n'importe qui de savoir s'il est susceptible d'être intéressé par une liste de diffusion, et aussi de permettre la recherche de questions déjà posées avant d'en poser une (typiquement pour la liste **foobar-users**).

5.2 Site Web

5.2.1 Buts

Il s'agit de mettre en place quelques pages web afin de rassembler l'information sur le logiciel, le présenter, le promouvoir, indiquer comment l'obtenir et où se trouve la documentation, etc.

5.2.2 Outils

Les pages web peuvent être créées grâce à n'importe quel outil disponible (emacs, éditeur de pages web spécialisé comme Amaya, etc.). Un wiki peut aussi être une façon de faire les pages du site. Enfin, on utilisera un outil de validation HTML/XML pour s'assurer de la correction des pages (voir par example le site du W3C www.w3c.org).

Un patron de site est disponible ici : devel.inria.fr/recom/files/template_site.tar.gz

Les sources de ce patron sont disponibles, vous pouvez les adapter à votre site : devel.inria.fr/recom/files/devel_recom_site.tar.gz

5.2.3 Pratiques

Le site du logiciel devrait mentionner l'INRIA et le(s) projet(s) au sein duquel ou desquels il a été développé, avec des liens vers leurs pages respectives.

Un site minimum devrait comporter les pages suivantes :

- **About / A propos** : Cette page doit présenter le logiciel, ce qu'il fait, comment il fonctionne (en gros), la license de distribution, éventuellement les publications relatives.
- **Downloads / Téléchargements** : Sur cette page doivent être indiqués les liens pour télécharger la ou les version(s) du logiciel, éventuellement pour plusieurs architectures et sous différentes formes (sources, paquetages, ...). Il est bon de donner pour chaque version (un lien vers) la liste des changements introduits et les corrections apportées par rapport à la version précédente.
- **Documentation** : Ici doivent se trouver les liens vers la ou les documentation(s), éventuellement dans différents formats. Il est recommandé, lorsque c'est possible, de proposer une version en ligne de la documentation, afin que le visiteur puisse avoir une idée de la qualité de celle-ci, ainsi que des versions téléchargeables pour une installation locale (un `.tar.gz` des pages web, par exemple). Des informations sur la documentation de logiciel : devel.inria.fr/wiki/themes/documenter
- **Frequently Asked Questions / Foire aux questions** : Il est recommandé d'enrichir au fur et à mesure cette page avec les questions (et leurs réponses) qui reviennent souvent. Il y a de grandes chances qu'un visiteur ne connaissant pas encore le logiciel se les pose lui aussi et appréciera d'avoir des réponses immédiatement, ce qui lui donnera une idée plus précise du logiciel, l'encourageant à l'essayer. De plus, cela lui évitera de faire perdre du temps aux développeurs en répondant encore une fois à l'une de ces questions.
- **Contacts** : Cette page doit indiquer qui sont les auteurs et comment les contacter, ainsi que les différentes listes de diffusion disponibles (cf. 5.1). Il est important également d'y indiquer l'adresse du gestionnaire de bogues éventuellement utilisé pour permettre aux utilisateurs de signaler des bogues qui seront automatiquement tracés (cf. 2.3).

Une fois le site mis en place, il faut annoncer et faire référencer le logiciel. Pour cela, consulter devel.inria.fr/wiki/themes/annoncer_logiciel.

5.3 Documentation technique

5.3.1 Buts

Maintenir une documentation technique à jour, pour les utilisateurs et les développeurs. Les développeurs ont besoin de savoir :

- comment utiliser l'environnement de développement de manière appropriée (comment compiler, comment gérer sa version expérimentale sans affecter le groupe, etc.),
- comment ajouter du code (contraintes d'architecture du logiciel, contraintes d'environnement ou d'installation),
- d'une manière générale, tout ce qui est utile pour contribuer efficacement au logiciel.

Les utilisateurs cherchent une documentation au sens traditionnel, c'est-à-dire un document qui va les aider à utiliser le logiciel :

- Manuel de référence d'une bibliothèque,
- Exemples d'utilisation (il est toujours apprécié de pouvoir copier-coller des exemples fonctionnels, qui couvrent les idiomes d'utilisation d'une bibliothèque).

5.3.2 Outils

Tous outils appropriés (Latex, Word, HTML, fichiers README, ...). Beaucoup utilisent Latex suivi d'Hevea, un traducteur Latex-HTML[4] D'autres utilisent des outils de "litterate programming", ou plus simplement des extracteurs de documentation (Javadoc, Doxygen).

5.3.3 Pratiques

Écrire une bonne documentation technique est difficile et consommateur de temps. Le strict minimum est de commenter le code. Nous recommandons (le cas échéant) une documentation des API maintenue en même temps que le code, par exemple à l'aide de Javadoc ou Doxygen, ainsi que des exemples commentés d'utilisation.

D'une manière générale, voici quelques astuces pour aider dans l'activité de rédaction :

- écrire la documentation d'une bibliothèque dès que l'API est fixée, au lieu d'attendre une semaine avant distribution,
- développer les tests comme exemples d'utilisation (une pierre deux coups),
- écrire la documentation du code d'un autre membre du projet lorsqu'on fait une revue de code.

5.4 Documentation utilisateur

5.4.1 Buts

Créer et maintenir à jour une documentation pour expliquer aux utilisateurs finaux comment utiliser le logiciel, sans qu'ils aient à en connaître le fonctionnement interne. Elle facilite ainsi la diffusion la plus large du logiciel.

Cette documentation doit bien entendu être adaptée au niveau des utilisateurs visés.

5.4.2 Outils

Différentes catégories de documentation utilisateur existent (e.g., tutoriaux, livres blancs, exemples d'utilisation, manuels de références), sous différents formats (e.g., papier, fichiers, en ligne).

[4] devel.inria.fr/wiki/logiciels/hevea

Il est hors du cadre de ce document de recommander des outils particuliers.

Soulignons cependant qu'il est capital de fournir la documentation pour les utilisateurs dans des **formats facilement lisibles** et largement accessibles, c'est-à-dire des formats courants, dont les outils de lecture sont disponibles sur différents systèmes d'exploitation, gratuits, voire souvent installés par défaut (ou au moins installables très facilement). Nous pouvons ainsi citer les formats **PDF et HTML**. Les outils (ou chaînes d'outils) choisis doivent donc permettre de produire des documents conformes à ces formats pratiques pour l'utilisateur final.

Remarquons qu'il est préférable d'utiliser des outils adaptés à une utilisation en édition collaborative et/ou avec un gestionnaire de versions (cf. section 2.2 page 10).

5.4.3 Pratiques

La documentation utilisateur se doit d'être particulièrement **claire et facile d'accès**. Ceci semble une trivialité, mais en pratique cet aspect n'est pas toujours bien considéré, alors qu'il peut contribuer sensiblement à la bonne (ou mauvaise) diffusion d'un logiciel.

Il est préférable de faire écrire conjointement cette documentation utilisateur par des développeurs et des non-développeurs, ou, au minimum, de la faire **relire par une personne extérieure** au développement. En effet, les développeurs plongés dans le code ont parfois du mal à prendre du recul et le point de vue des utilisateurs. L'utilisateur "cobaye" permet donc souvent de lever des points peu explicites pour l'utilisateur neophyte, qui ne seraient peut-être pas venu à l'esprit d'une personne connaissant mieux le logiciel.

Il est souhaitable de **commencer la documentation dès le début** du projet, à partir des objectifs et/ou des spécifications. Ceci permet par exemple une prise en main du logiciel par des utilisateurs le plus tôt possible dans le processus de développement, d'où un retour plus précoce contribuant à l'amélioration du logiciel lui-même.

En fonction du niveau d'expertise des utilisateurs prévus et du niveau de difficulté d'accès au logiciel, **différents types de documentation** utilisateur peuvent être prévus. Les exemples d'utilisation et les tutoriaux sont ainsi, par leur nature séquentielle, plus progressifs qu'un manuel de référence. Ce dernier est cependant extrêmement utile pour un accès direct pour un utilisateur expérimenté qui a un besoin ponctuel.

Il est souhaitable que la documentation soit **accessible depuis le logiciel lui-même** (ce qui n'exclut pas une version papier à côté) et ainsi toujours disponible, au minimum dans une version abrégée.

Enfin, rappelons l'importance de **maintenir la documentation à jour**, tâche fastidieuse mais indispensable, sous peine de perdre tout le bénéfice du travail passé et de nuire à la diffusion du logiciel.

Annexe : méthode de travail

Pour rédiger ce document, nous avons choisi la méthode de travail suivante :

1. Connaître nos pratiques internes par un état des lieux des projets développant du logiciel, en particulier lorsque ce logiciel est diffusé (les prototypes de recherche concernant un seul thésard ou chercheur sont moins prioritaires).
2. Identifier les projets ayant le plus de maturité, sur l'ensemble du processus ou sur un aspect particulier.

3. Dégager un éventail de pratiques et outils que nous pouvons recommander, c'est-à-dire dont nous pouvons montrer le succès dans un autre projet de recherche INRIA et dont nous pouvons soutenir.
4. Faire partager ces pratiques et outils pour augmenter le niveau minimum de maturité dans une majorité des projets INRIA diffusant du logiciel.

L'étape 1 est relativement simple à exécuter, mais elle prendra du temps ; il faut que chaque cellule de support fasse le recensement des développements dans les projets, dès lors qu'un logiciel est diffusé (même si la diffusion est limitée à une communauté scientifique restreinte). A noter également que les cellules de support restent en contact permanent avec les projets grâce aux "correspondants développement logiciels", ce qui permettra de mettre à jour notre connaissance des pratiques des projets.

L'étape 2 est la conséquence naturelle de 1. Comme il semble difficile de parler dans l'abstrait de processus de développement aux chercheurs, comme il semble également difficile et inadéquat de citer les méthodes pratiquées dans l'industrie de l'édition du logiciel où l'environnement est différent, nous pensons qu'une méthode plus adaptée à notre culture de recherche est de prendre exemple sur les pratiques des projets les plus expérimentés.

L'étape 3 est un travail de synthèse qui est commencé ici. Pour illustrer le succès de telle pratique ou de tel outil, nous pourrons (si les ressources le permettent) récolter des "success stories".

L'étape 4, la plus délicate, relève du travail de différents acteurs : un groupe de travail qui définit les éléments du processus de développement ; les services de support au développement logiciel qui évangélisent les projets et fournissent du support ; les projets eux-mêmes qui décident (ou non) de participer au programme. Ce travail s'inscrit dans la durée : les équipes vont (normalement) progresser en maturité ; notre définition d'un processus de développement va (normalement) évoluer avec le temps, en couverture (plus d'éléments de processus) ou en profondeur (plus de détails dans les méthodes).