Karl Fogel

Produire du Logiciel libre

Framabook

Karl Fogel

Produire du logiciel libre

Traduction Fr. : Framalang

Framabook
le pari du livre libre

2010 : traduction Framasoft (projet Framalang)

Mise en page : La Poule ou l'Œuf

Couverture : Neocrea

Dépôt légal : décembre 2010, In Libro Veritas

ISBN : 978-2-35922-034-6

ILV-BIBLIOTHECA
ÉDITIONS

Avant-propos

Nous sommes heureux de publier cette traduction française de *Producing Open Source Software. How to Run a Successful Free Software Project*, dont le texte original est accessible sur ProducingOSS.com [1], dans de nombreuses autres langues.

Pour les habitués du genre, la plupart des *How To* — ou « comment faire… » — en informatique sont des compléments aux manuels de logiciels. En se focalisant sur des tâches spécifiques, un *How to* permet de montrer, par la pratique, comment se servir de telle ou telle fonctionnalité d'un programme. Dès lors, comment concevoir qu'il puisse exister un *How to* censé renseigner sur une activité éminemment sociale comme celle qui consiste à produire du logiciel libre par une stratégie de projet communautaire regroupant des programmeurs bénévoles ?

Grâce à son expérience du développement Open Source, Karl Fogel nous livre ici bien davantage qu'une simple marche à suivre pour qu'un projet voie le jour et ait une chance d'aboutir. Il s'agit en effet de détailler les éléments stratégiques les plus importants comme la bonne pratique du courrier électronique et le choix du gestionnaire de versions, mais aussi la manière de rendre cohérents et harmonieux les rapports humains tout en

1. http ://producingoss.com/

ménageant les susceptibilités… En somme, dans le développement Open Source peut-être plus qu'ailleurs, et parce qu'il s'agit de trouver un bon équilibre entre coopération et collaboration, les qualités humaines sont aussi décisives que les compétences techniques.

La traduction de cet ouvrage a obéi aux mêmes principes que ceux exposés par Karl Fogel. Ce sont les mêmes enjeux de motivation et de disponibilité des participants que soulève un tel projet, et qui expliquent très certainement sa relative lenteur, puisque les travaux commencèrent au début de l'année 2007. C'est en revanche toute la force des projets libres de Framasoft que de ne pas être soumis à une exigence de rentabilité imposée par des frais d'investissement ou par les contraintes de la concurrence, si sévères dans le monde éditorial. Aussi, quel que soit son rythme, la réalisation d'une traduction et sa publication dans la collection Framabook est toujours vouée au succès, mais à plus ou moins long terme.

Produire du logiciel libre est d'abord le résultat d'une convergence fortuite entre deux projets isolés, comme c'est bien souvent le cas dans le monde du libre. Au cours de l'année 2007, Bertrand Florat et Étienne Savard avaient commencé à traduire l'ouvrage selon les spécifications données par Karl Fogel : un Docbook sous SVN, où les contributeurs volontaires pouvaient reporter leurs « modifications » à partir du texte en anglais sur le site officiel ProducingOSS.com. Le groupe Framalang, de son côté, avait depuis quelque temps placé *Producing Open Source Software* dans la liste des textes à traduire sur le wiki consacré [1], avec un responsable : Olivier Rosseler. Comme l'explique Johann Bulteau, témoin et acteur de la première heure :

> *« C'est vers le mois de juin 2007 qu'Eva Mathieu, de l'April [2], et qui travaillait sur Framalang, fit suivre à Pierre-Yves Gosset [3] un courriel qu'elle avait reçu 'par erreur', car les auteurs voulaient contacter les responsables de Framalang. Pierre-Yves les renvoya donc vers nous.*
>
> *Ils s'agissait d'Étienne Savard et de Bertrand Florat, les deux traducteurs 'officiels' de ce bouquin. Quand je*

1. http ://www.framalang.org/
2. http ://www.april.org/
3. Pierre-Yves Gosset est le Délégué Général (et permanent) de Framasoft.

dis ‘officiels’, c’est parce qu’ils étaient en contact avec l’auteur, Karl Fogel, et que leur travail se passait directement sur ProducingOSS.com.

Il y eut moult discussions par courriel afin de s’accorder sur la manière de procéder : fork ou coopération ? Si coopération, où et comment travailler ? Ce qui était déjà sûr en tout cas, c’est que Karl Fogel était ravi de voir que son livre allait être traduit en français !

On a discuté des méthodes de travail. Eux fonctionnaient directement sur SVN, avec export en Docbook. Le problème avec ça, c’est que nos traducteurs ne sont pas tous des informaticiens chevronnés. Ils n’étaient pas là pour et surtout ne savaient pas faire des commits de leur travail… Utiliser ces outils aurait donc fait perdre une grosse partie des forces vives. Ils l’ont compris et ils ont finalement décidé de rejoindre notre wiki. »

Pour Olivier Rosseler, il s’agissait surtout d’un défi. Le groupe de traduction Framalang avait pris naissance grâce à la traduction du livre *Changer pour OpenOffice.org*, et la perspective d’une nouvelle publication était fortement motivante. Renouveler l’opération signifiait qu’un projet permanent de traduction collaborative était non seulement viable mais constituait une ressource très importante pour la diffusion des connaissances sur le logiciel libre. D’autres projets vinrent alors se greffer sur les priorités initiales, à commencer par le projet de traduction de *Free as in Freedom*, la biographie de Richard Stallman par Sam Williams [1].

Néanmoins, en tant qu’une des principales portes d’entrée francophone sur le logiciel libre, Framasoft se voyait conforté dans le choix du livre de Karl Fogel, car il représentait une chance de diffuser aussi une méthode pratique de production de logiciel libre, c’est-à-dire un modèle de développement, d’économie et d’innovation encore trop méconnu, au moins en France.

Selon Olivier Rosseler, la traduction elle-même a suivi les étapes de développement mentionnées par Karl Fogel :

1. Cette biographie, d’abord traduite, fut finalement réécrite dans le cadre d’une collaboration entre Richard Stallman et le projet Framabook, aux éditions Eyrolles (2010) : *Richard Stallman et la Révolution du logiciel libre. Une biographie autorisée*.

> *« Curieusement, les propos de Karl Fogel dans son livre se sont vérifiés tout au long de notre traduction, comme si, à maintes reprises, il décrivait précisément notre collaboration : le besoin à l'origine, les relations si particulières entre des contributeurs qui ne sont liés que par le projet, mais qui ont paradoxalement des objectifs différents, les périodes d'activité creuses qui alternent avec la frénésie du contributeur pris par les remords d'avoir délaissé le projet, l'approche de la publication et l'activité qu'elle génère…*
>
> *J'en retiendrai qu'un projet libre doit effectivement naître d'un réel besoin. Si personne n'a vraiment intérêt à ce que le projet voie le jour, personne n'est là pour vraiment en prendre les rênes. »*

En dernier lieu, c'est l'équipe Framabook [1] qui se chargea de la mise en œuvre de la présente publication. Les phases de relecture, de reformulation et de chasse aux coquilles furent elles-mêmes menées à plusieurs en utilisant notre chaîne éditoriale La Poule ou l'Œuf [2], que nous tenons à vivement remercier ici. L'efficacité de cette ultime étape collaborative fut impressionnante, mais nous n'aurions pas la prétention de qualifier ce travail d'exemplaire. C'est pourquoi nous avons toujours besoin des « rapports de bogues » de la part des lecteurs, et pour cela, n'hésitez pas à vous rendre sur le wiki de Framasoft [3] pour contribuer à ce Framabook.

1. http ://www.framabook.org
2. http ://www.pouleouoeuf.org/
3. http ://wiki.framasoft.org/Contribuer_aux_Framabooks

Remerciements

Framasoft tient à remercier Framalang, son équipe de traduction, et plus particulièrement, pour la traduction et la relecture initiale :

– Johann Bulteau
– Claude
– Cœurgan
– Daria
– Bertrand Florat
– Gaelix
– Benjamin Jean (alias Mben)
– Eva Mathieu
– R4f
– Olivier Rosseler
– Étienne Savard
– Tolosano
– Tyah

Puis pour la reprise, relecture et mise en page de l'ouvrage avant publication :

– Barbara Bourdelles (alias Garburst)
– Gilles Coulais (alias Poupoul2)
– Christophe Jarry
– Jean-Bernard Marcon (alias Goofy)
– Christophe Masutti
– Julien Reitzel
– Raymond Rochedieu (alias Papiray)
– Frédéric Urbain

Remerciements également à Simon Descarpentries (alias Siltaar) pour la coordination du projet de publication.

Enfin, Framasoft remercie tout particulièrement Christophe Masutti, coordinateur de la collection « Framabook ».

Remerciements de l'auteur

Ce livre est dédié à deux amis qui me sont chers et sans qui rien de tout cela n'aurait été possible : Karen Underhill et Jim Blandy.

Préface

Pourquoi écrire ce livre ?

Dans les soirées, les gens ne me regardent plus bizarrement quand je leur dis que je fais du logiciel libre. « Ah oui, Open Source, comme Linux ? » me répondent-ils. J'acquiesce avec empressement. « Oui, exactement, c'est ce que je fais ! » C'est agréable de ne plus être complètement à l'écart. Dans le passé, la question suivante était généralement assez prévisible : « Comment gagnes-tu ta vie en faisant ça ? ». Ma réponse résumait toute l'économie de l'Open Source : certaines organisations trouvent leur intérêt dans l'existence d'un logiciel particulier, mais n'ont aucunement besoin d'en vendre des copies, elles veulent juste être sûres que le logiciel continue d'exister et d'être maintenu : en tant qu'outil plutôt que bien.

Ces derniers temps pourtant, la question ne portait pas toujours sur l'argent. Le côté économique de l'Open Source [1] n'est plus si mystérieux et beaucoup de non-programmeurs comprennent, ou du moins ne sont pas surpris, que certains y soient employés à plein temps. Par contre, la question entendue de plus en plus souvent était : « Ah ! Comment ça marche ? »

1. Les termes « Open Source » et « libre » sont essentiellement des synonymes dans ce contexte : j'en parle davantage dans la section intitulée « 'Libre' contre 'Open Source' » dans l'Introduction.

Je n'avais pas de réponse satisfaisante toute prête et, plus j'essayais d'en donner une, plus je réalisais à quel point le sujet est complexe. Mener un projet de logiciel libre n'est pas exactement comme diriger une entreprise (imaginez devoir constamment discuter de la nature du produit avec un groupe de volontaires que vous ne rencontrerez jamais pour la plupart !). Pour différentes raisons, ce n'est pas non plus comme diriger une association à but non lucratif traditionnelle ou un gouvernement. Il y a des ressemblances entre tous ces types d'organisations, mais je suis lentement arrivé à la conclusion que le logiciel libre est *sui generis*. Il peut être comparé à de nombreuses choses, sans pouvoir être assimilé à aucune. En fait, même l'hypothèse selon laquelle un projet de logiciel libre peut être « dirigé » est douteuse. Un tel projet peut être démarré et influencé par les personnes impliquées, souvent même très fortement. Mais son capital ne peut être considéré comme la propriété d'un seul, et, tant qu'il y aura des volontaires, quelque part et peu importe où, nul ne pourra décider unilatéralement de l'arrêter. Chacun possède un pouvoir infini et personne ne possède le moindre pouvoir : le tout produisant une dynamique intéressante.

Voilà pourquoi j'ai voulu écrire ce livre. Les projets de logiciels libres ont donné naissance à une culture distincte, une philosophie où la liberté de créer un logiciel réalisant ce que l'on veut est la doctrine centrale. Pourtant, il résulte de cette liberté non pas une dispersion des individus, chacun allant dans sa propre direction avec le code, mais une collaboration enthousiaste. En effet, cette capacité est l'une des facultés la plus hautement estimée dans le monde du libre. Diriger de tels projets signifie s'engager dans une sorte de coopération hypertrophiée où l'habileté d'une personne non seulement à travailler avec les autres, mais aussi à proposer de nouvelles méthodes de travail en commun, peut aboutir à des avancées tangibles du logiciel. Ce livre tente d'expliquer comment rendre ceci possible. Il n'est en aucun cas exhaustif, mais peut être considéré comme une base de départ.

Produire du bon logiciel libre est en soi un but valable, et j'espère que les lecteurs qui viennent chercher des moyens de l'atteindre seront satisfaits de ce qu'ils trouveront ici. Bien au-delà, j'espère transmettre un peu du grand plaisir que l'on peut prendre à travailler avec une équipe motivée de développeurs Open Source et à interagir avec les utilisateurs de manière

directe, comme le monde du libre nous y invite de façon géniale. Prendre part à un projet de logiciel libre réussi est amusant et c'est bien ce qui, au bout du compte, permet au système entier de fonctionner.

À qui s'adresse ce livre ?

Ce livre s'adresse aux développeurs de logiciels et aux responsables envisageant de lancer un projet Open Source, ou l'ayant déjà commencé, et qui se demandent que faire ensuite. Il devrait également être utile à ceux qui souhaitent simplement s'impliquer dans un projet Open Source sans l'avoir jamais fait auparavant.

Nul besoin d'être programmeur, mais le lecteur devrait connaître les concepts basiques d'ingénierie logicielle tels que *code source*, *compilateur* et *correctif*.

Une expérience préalable du logiciel Open Source, en tant qu'utilisateur ou développeur, n'est pas nécessaire. Ceux qui ont déjà travaillé au sein d'un projet de logiciel libre trouveront certaines parties du livre évidentes et pourront les passer. Étant donné le large éventail possible d'expériences des lecteurs, j'ai essayé de nommer clairement les différentes sections et de préciser celles qui peuvent être ignorées par les familiers du sujet.

Sources

Une grande partie de la matière première de ce livre provient de cinq ans d'expérience sur le projet Subversion [1], un programme Open Source de gestion de versions, écrit à partir de rien, et créé pour remplacer CVS, lui-même une référence en la matière dans la communauté Open Source. Le projet fut lancé, au début des années 2000, par mon employeur CollabNet [2] lequel comprit, heureusement dès le départ, comment le faire fonctionner de manière collective et répartie. De nombreux développeurs bénévoles y adhérèrent très tôt. Aujourd'hui, le projet en compte une cinquantaine dont peu sont employés par CollabNet.

1. http ://subversion.tigris.org/
2. http ://www.collab.net/

De bien des manières, Subversion est un exemple classique de projet Open Source et je m'en suis finalement inspiré plus que je ne le pensais au début, en partie parce que c'était pratique pour moi. Chaque fois que j'avais besoin d'un exemple pour illustrer un phénomène particulier, c'est une expérience issue de Subversion qui me venait à l'esprit. Mais aussi pour une question de vérification : quoiqu'engagé, à divers degrés, dans d'autres projets de logiciels libres et malgré mes entretiens avec des amis et connaissances impliqués dans bien d'autres encore, j'ai réalisé rapidement, tout en écrivant en vue d'une publication, que tous les faits doivent être vérifiés. Je ne voulais pas faire de déclarations à propos d'évènements se produisant dans d'autres projets en me basant simplement sur ce que je pouvais lire dans les archives de leurs listes de diffusion publique. Si quelqu'un le faisait avec Subversion, je m'en rends bien compte, il se tromperait la moitié du temps. Donc quand j'ai pris exemple sur d' autres projets avec lesquels je n'ai pas eu d'expérience directe, j'ai toujours essayé de vérifier mes informations auprès d'une personne impliquée, en qui je pouvais avoir confiance, afin qu'elle m'explique ce qui s'est vraiment passé.

J'ai travaillé sur Subversion pendant les cinq dernières années, mais je suis impliqué dans le logiciel libre depuis douze ans. Voici d'autres projets ayant influencé ce livre :

- Le projet d'éditeur de texte GNU Emacs de la Free Software Foundation, pour lequel je maintiens quelques paquets.
- Concurrent Versions System (CVS), sur lequel j'ai beaucoup travaillé en 1994-1995 avec Jim Blandy mais auquel je participe seulement par intermittence depuis.
- L'ensemble des projets Open Source connus sous le nom d'Apache Software Foundation et plus spécialement l'Apache Portable Runtime (APR) et l'Apache HTTP Server.
- OpenOffice.org, la base de données de Berkeley de Sleepycat et la base de données MySQL : je n'ai pas été personnellement impliqué dans ces projets mais je les ai observés et, dans certains cas, j'ai pu parler avec des personnes en faisant partie.
- Le GNU Debugger (GDB) (idem).
- Le projet Debian (idem).

Évidemment, cette liste n'est pas exhaustive. Comme la plupart des programmeurs Open Source, je garde un œil sur de nombreux projets différents, juste pour avoir une idée générale de ce qui se passe. Je ne vais pas tous les mentionner ici, mais certains sont cités dans le livre quand je l'ai jugé approprié.

Remerciements

Écrire ce livre m'a pris quatre fois plus de temps que je ne le pensais et, quel que soit le sujet, j'avais souvent l'impression de marcher avec une épée de Damoclès suspendue au-dessus de ma tête. Sans l'aide de nombreuses personnes, j'aurais été incapable de le finir en gardant tous mes esprits.

Andy Oram, chez O'Reilly, a été pour moi l'éditeur idéal. En plus de très bien connaître le sujet (il a suggéré un grand nombre des thèmes), il a le don rare de savoir ce que quelqu'un veut dire et de pouvoir l'aider à le dire correctement. Travailler avec lui a été un honneur. Merci aussi à Chuck Toporek pour avoir immédiatement proposé ce projet à Andy.

Brian Fitzpatrick a relu tout mon travail à mesure que j'écrivais, ce qui a non seulement amélioré le livre mais m'a aussi permis de poursuivre l'écriture quand j'aurais souhaité me trouver n'importe où plutôt que devant mon ordinateur. Ben Collins-Sussman et Mike Pilato ont aussi gardé un œil sur l'avancement du travail et étaient toujours heureux de discuter, parfois longuement, quel que soit le sujet que j'essayais de traiter cette semaine-là. Ils remarquaient également quand j'avais tendance à me relâcher et me taquinaient si nécessaire. Merci les gars !

Biella Coleman rédigeait également un mémoire en même temps que j'écrivais ce livre. Elle sait ce que signifie s'asseoir et écrire tous les jours : elle fut pour moi aussi bien un exemple qu'une oreille compatissante. Elle possède aussi le regard fascinant de l'anthropologue sur le mouvement du logiciel libre, me fournissant à la fois des idées et des références que je pouvais utiliser. Alex Golub (un autre anthropologue, avec un pied dans le monde du logiciel libre, qui lui aussi terminait son mémoire au même moment) m'a apporté un soutien exceptionnel dès le début, ce qui m'a beaucoup aidé.

Micah Anderson n'a, d'une certaine manière, jamais semblé trop stressé par son propre contrat d'édition, ce qui fut source de motivation mais me rendait maladivement jaloux. Il a toutefois toujours été présent par son amitié, sa conversation et (au moins à une occasion) son aide technique. Merci Micah !

Jon Trowbridge et Sander Striker m'ont apporté chacun leurs encouragements et une aide précieuse. Leur grande expérience du logiciel libre m'a fourni la matière que je n'aurais pu trouver nulle part ailleurs.

Merci à Greg Stein, non seulement pour son amitié et ses encouragements arrivés à point nommé, mais aussi pour avoir montré au projet Subversion à quel point une inspection régulière du code est importante pour le développement d'une communauté de programmeurs. Je remercie également Brian Behlendorf qui a fait entrer avec tact dans nos esprits l'importance des discussions publiques : j'espère que ce principe se reflète dans ce livre.

Merci à Benjamin « Mako » Hill et Seth Schoen pour les nombreuses discussions à propos du logiciel libre et de sa politique, à Zack Urlocker et Louis Suarez-Potts pour avoir trouvé quelques minutes dans leur emploi du temps surchargé afin de m'accorder une interview, à Shane de la liste Slashcode pour m'avoir permis de citer son article et à Haggen So pour sa comparaison des sites d'hébergement qui m'a énormément aidé.

Merci à Alla Dekhtyar, Polina et Sonya pour leurs encouragements patients et sans faille. Je suis ravi de ne plus avoir à écourter (ou plutôt, essayer en vain d'écourter) nos soirées pour aller travailler sur « Le Livre ».

Merci à Jack Repenning pour son amitié, sa conversation et son entêtement à refuser d'accepter une analyse simple mais fausse quand une autre plus complexe mais juste existe. J'espère qu'une partie de sa longue expérience dans le développement et l'industrie du logiciel a déteint sur ce livre.

CollabNet s'est montré exceptionnellement généreux en me permettant d'adapter mon agenda pour écrire ce livre, sans se plaindre quand cela me prit plus de temps que prévu. Je ne connais pas tous les rouages d'une telle décision mais je suppose que Sandhya Klute, et par la suite Mahesh Murthy, y sont pour quelque chose. Je les remercie tous les deux.

Toute l'équipe de développement Subversion a été une source d'inspiration au cours de ces cinq dernières années et c'est auprès de ses membres que j'ai appris l'essentiel de ce qui est expliqué dans ce livre. Je ne les remercierai pas nominativement, car ils sont trop nombreux, mais j'implore chaque lecteur qui croise un *committer* de Subversion de lui offrir un verre. C'est ce que je compte faire moi-même.

J'ai souvent pesté auprès de Rachel Scollon à propos de l'état de ce livre. Elle a toujours été prête à m'écouter et, d'une certaine manière, a réussi à faire paraître mes problèmes moins graves qu'ils ne l'étaient avant nos conversations. Cela m'a beaucoup aidé, merci.

Merci (encore) à Noel Taylor, qui a certainement dû se demander pourquoi je voulais écrire un autre livre, sachant à quel point je m'étais plaint la fois précédente, mais dont l'amitié et la conduite m'ont aidé à préserver, dans mon existence, la musique et un cercle fraternel, même pendant les périodes les plus chargées. Merci aussi à Matthew Dean et Dorothea Samtleben, amis et compagnons musicaux que j'ai longtemps fait souffrir et qui se sont montrés très compréhensifs quand les excuses que je donnais pour ne pas répéter s'accumulaient. Megan Jennings m'a toujours soutenu et a montré un réel intérêt pour le sujet malgré son inexpérience dans le domaine, un vrai énergisant pour un écrivain qui doute. Merci mon amie !

J'ai eu quatre relecteurs compétents et persévérants pour ce livre : Yoav Shapira, Andrew Stellman, Davanum Srinivas et Ben Hyde. Si j'avais pu ajouter toutes leurs excellentes suggestions, ce livre serait encore meilleur. Des contraintes de temps m'ont forcé à faire des choix mais les améliorations sont quand même visibles. Toutes les erreurs qui subsistent sont entièrement les miennes.

Mes parents, Frances et Henry, m'ont apporté comme toujours un magnifique soutien, et comme ce livre est moins technique que le précédent, j'espère qu'ils le trouveront un peu plus lisible.

Pour finir, j'aimerais remercier les dédicataires : Karen Underhill et Jim Blandy. L'amitié et la compréhension de Karen représentent tout pour moi, non seulement pendant l'écriture de ce livre mais aussi au cours des sept dernières années. Je n'aurais simplement pas pu finir sans son aide. De même, Jim, véritable ami et maître hacker qui, le premier, m'a fait découvrir les logiciels libres tel l'oiseau enseignant à l'avion comment voler.

Note

Les pensées et opinions exprimées dans ce livre sont les miennes. Elles ne représentent pas nécessairement les idées de CollabNet ni du projet Subversion.

Table des matières

Avant-propos **iii**

Préface **viii**

Introduction **1**

Bien d''marrer **15**
1. Commencez avec ce que vous avez 17
2. Choisir une licence et l'appliquer 29
3. Donner le ton 32
4. Annoncer 39

Infrastructure technique **43**
1. Les besoins d'un projet 45
2. Les listes de diffusion 45
3. Les logiciels de gestion de versions 56
4. Suivi de bogues 68
5. IRC et les chats en temps réel 73
6. Les flux RSS 76
7. Les wikis 76
8. Les sites Web 78

Infrastructure sociale et politique **81**
1. Fourchabilité 82
2. Les dictateurs bienveillants 83
3. La démocratie basée sur le consensus 84
4. Tout mettre par écrit 90

L'argent **93**
1. Les différentes participations 95
2. Recrutez pour le long terme 96
3. Montrez-vous unis 97
4. Ne cachez pas vos motivations 98
5. L'argent ne fait pas tout 100
6. Signer des contrats 101
7. Financer ce qui ne touche pas à la programmation 105
8. Marketing 109

Communication **113**
1. Vous êtes ce que vous écrivez 114
2. Éviter les pièges courants 122
3. Les personnes difficiles 129
4. Gérer la croissance 132
5. Pas de discussion dans le système de suivi de bogues 139
6. Les annonces 141

Paquets, sorties et développement quotidien **151**
1. Numérotation des versions 152
2. Les branches de publication 157
3. Stabiliser une version 160
4. La création de paquets 164
5. Tests et sorties 170
6. Maintenir plusieurs versions 172
7. Publications et développement quotidien 173

Encadrer les volontaires **177**
1. Tirer le meilleur des volontaires 178
2. Partager les tâches 190
3. Transitions 197
4. *Committers* 200
5. Remerciements 204
6. Les fourches 206

Licences, droits d'auteur et brevets **211**
1. Terminologie 211
2. Les aspects des licences 214
3. La GPL et la compatibilité de licence 216
4. Le choix d'une licence 217
5. Attribution des droits d'auteur et propriété 221
6. La double licence 224
7. Brevets 225
8. Plus de références 228

A Solutions libres de gestion de versions **231**

B Systèmes libres de suivi de bogues **237**

C Pourquoi se soucier de la couleur de l'abri à vélos **241**

D Exemples d'instructions pour les rapports de bogues **247**

E Copyright **251**

Introduction

La plupart des projets de logiciels libres échouent.

Nous avons tendance à ne pas trop remarquer les échecs. Seuls les succès attirent notre attention, et le nombre total [1] de projets de logiciels libres est si important que leur visibilité reste forte malgré un taux de réussite relativement faible. De même, nous n'entendons pas parler des échecs car l'insuccès est un non-évènement. Le moment précis où un projet cesse d'être viable n'existe pas : les gens s'en écartent et finissent par l'abandonner. Il se peut qu'à un moment donné un dernier changement soit apporté au projet mais l'auteur, à cet instant, ne sait pas qu'il s'agit du dernier. Comment déterminer la mort d'un projet ? Quand on n'y a plus travaillé activement depuis six mois ? Quand le nombre de ses utilisateurs cesse de croître, sans avoir dépassé celui des développeurs ? Et que dire d'un projet abandonné parce que ses développeurs, se rendant compte qu'ils reproduisaient un travail existant, se décident à rejoindre un autre projet pour l'améliorer en y intégrant une grande partie de leurs travaux précédents ? Le premier projet est-il mort ou a-t-il simplement changé d'adresse ?

En raison de cette complexité, il est impossible de déterminer précisément le taux d'échec. Mais le constat qui ressort de plus d'une décennie dans l'Open Source, de quelques participations à SourceForge.net et d'un peu de « googlage » est le suivant : ce taux est extrêmement élevé, probablement de l'ordre de 90% à 95%. Il l'est encore plus si l'on y inclut les projets survivants mais qui fonctionnent mal : certes ils produisent des applications utilisables, mais il est pénible d'y travailler, ou bien il est impossible d'y progresser comme on pourrait le souhaiter.

1. SourceForge.net est un site d'hébergement populaire qui accueillait 79 225 projets enregistrés à la mi-avril 2004. On est bien sûr très loin de la quantité totale de projets de logiciels libres sur Internet : il s'agit seulement ici de ceux qui ont choisi d'utiliser SourceForge.

L'objectif de cet ouvrage est d'éviter l'échec. Il passe en revue non seulement les bonnes pratiques, mais aussi les mauvaises, afin que chacun puisse détecter et corriger les problèmes rapidement. Mon plus grand souhait est qu'après sa lecture, on puisse disposer d'un répertoire de techniques pour, d'une part, éviter les pièges les plus courants du développement Open Source, et d'autre part, gérer la croissance et la maintenance d'un projet réussi. Le succès n'est pas un jeu à somme nulle et il ne s'agit pas ici de victoire ni de domination de la concurrence. En réalité, une part importante du développement d'un projet Open Source consiste à travailler sans heurt avec les autres projets apparentés. À long terme, chaque réussite contribue au bien-être du « logiciel libre », dans son ensemble et partout dans le monde.

Il serait tentant de dire que les projets de logiciels libres et les projets à caractère propriétaire échouent pour les mêmes raisons. Le libre n'a pas le monopole des agendas farfelus, des spécifications floues, de la mauvaise gestion des ressources humaines, des phases de conception insuffisantes et autres nuisances bien connues de l'industrie du logiciel. La somme d'écrits traitant de ce sujet est déjà considérable, je ne m'étendrai donc pas davantage. J'essaierai plutôt de décrire les problèmes spécifiques au logiciel libre. Quand un projet s'écroule c'est souvent parce que les développeurs (ou les directeurs) n'ont pas su évaluer les problèmes propres au développement d'un logiciel Open Source, alors même qu'ils étaient rodés aux difficultés mieux connues du développement à code fermé.

Mesurez vos attentes vis-à-vis de l'Open Source, ne tombez pas dans le piège d'une trop grande ambition. Une licence ouverte ne met pas immédiatement des légions de développeurs au service de votre projet, et ouvrir le code d'un projet en difficulté n'est pas un remède à tous les maux. En fait, c'est plutôt le contraire : ouvrir un projet peut ajouter une nouvelle série de complications et coûte plus cher à court terme que le garder fermé. Ouvrir veut dire remanier le code pour le rendre compréhensible par quelqu'un de complètement étranger au projet, mettre en place un espace Web de développement, des listes de diffusion et, souvent, écrire pour la première fois la documentation. Tout ceci représente beaucoup de travail. Bien sûr, si des développeurs intéressés se présentent, il faudra les intégrer, répondre à leurs questions, tout cela peut prendre un certain temps avant que vous ne perceviez les bénéfices de leur présence. Comme l'a dit Jamie Zawinski en parlant des périodes troubles du début du projet Mozilla :

> *« L'Open Source fonctionne, mais ce n'est vraiment pas la panacée. La morale de cette histoire c'est qu'on ne peut pas prendre un projet moribond et le traiter à la poudre de perlimpinpin « Open Source » pour que tout se mette à marcher comme par magie. Le développement logiciel c'est difficile. Les solutions ne sont pas si simples. »* [1]

Lésiner sur la présentation et la création de paquets en remettant cela à plus tard, une fois le projet sur les rails, est une erreur. La présentation et la création de paquets comportent un nombre important de tâches visant à en faciliter l'approche. Pour rendre le projet accueillant aux néophytes, il faut écrire la documentation développeur et utilisateur, créer pour le projet un site Web informant les curieux, automatiser autant que possible la compilation et l'installation du logiciel, etc. Beaucoup de programmeurs considèrent malheureusement ce travail

1. Voir Jamie Zawinski, « Resignation and Postmortem », 1999. Article accessible à http ://www.jwz.org/gruntle/nomo.html.

comme secondaire par rapport au code lui-même. Et ceci pour plusieurs raisons. Premièrement, c'est à leurs yeux une perte de temps car ils n'en tirent pas directement les bénéfices contrairement aux personnes moins familières avec le projet. Après tout, les gens qui développent le projet n'ont pas vraiment besoin qu'on leur prépare des paquets. Ils savent déjà comment installer, administrer et utiliser le logiciel qu'ils ont écrit. Deuxièmement, les compétences requises pour la présentation et la création de paquets sont souvent complètement différentes de celles nécessaires à l'écriture du code. Les gens ont tendance à se concentrer sur ce qu'ils connaissent le mieux, même s'ils peuvent rendre un meilleur service au projet en consacrant un peu de temps aux choses qui leur conviennent moins. Le chapitre 2 examine en détail les questions de présentation et de création de paquets. Il explique pourquoi il est essentiel d'en faire une priorité dès le lancement du projet.

Vient ensuite l'idée fausse que l'Open Source n'a guère besoin de gestion de projet et qu'inversement les techniques de direction utilisées pour les développements en interne fonctionneront également pour le projet Open Source. Le management dans un projet Open Source n'est pas toujours très visible, mais dans les projets réussis il est toujours présent en coulisse d'une manière ou d'une autre. Inutile de pousser la réflexion très loin pour s'en rendre compte. Un projet Open Source consiste en un assemblage fortuit de programmeurs, catégorie déjà réputée pour son indépendance d'esprit, qui ne se sont très probablement jamais rencontrés et qui peuvent participer en ayant chacun des objectifs personnels différents. Il est facile d'imaginer ce que deviendrait un tel groupe *sans* management. À moins d'un miracle, il s'écroulerait ou éclaterait très vite. Les choses ne vont pas marcher d'elles-mêmes, que nous le voulions ou non. Mais le management, aussi actif soit-il, est le plus souvent informel, subtil et voilé. La seule chose qui maintienne un groupe de développeurs ensemble est la croyance commune qu'ils peuvent faire plus collectivement qu'individuellement. Dans ce cadre, le management a pour but principal de faire en sorte qu'ils continuent à le croire, en établissant des formes de communication, en s'assurant que des développeurs utiles ne sont pas marginalisés en raisons de caractéristiques personnelles et, de manière générale, en faisant en sorte que le projet reste un espace où les développeurs ont envie de revenir. Les techniques spécifiques pour réussir cela sont abordées dans le reste de l'ouvrage.

Enfin, il y a une catégorie générique de problèmes qu'on pourrait appeler « échecs de navigation culturelle ». Il y a dix ou même cinq ans, il aurait été prématuré de parler d'une culture mondiale du logiciel libre : plus maintenant. Une culture reconnaissable a émergé lentement et bien qu'elle ne soit pas monolithique (elle est autant sujette à la dissidence et aux factions que n'importe quelle culture géographiquement définie), elle est basée sur un noyau fondamentalement solide. La plupart des projets Open Source réussis affichent quelques-unes sinon toutes les caractéristiques de ce noyau. Ils récompensent certains types de comportements et en punissent d'autres, ils créent une atmosphère qui encourage la participation non planifiée (parfois au détriment de la coordination centralisée), ils possèdent leur propre conception de la politesse et de la brutalité pouvant différer foncièrement de celle qui prévaut par ailleurs. Et, chose primordiale, les participants de longue date ont généralement intériorisé ces critères au point d'être plus ou moins unanimes sur les comportements attendus. Les projets qui échouent généralement s'écartent de manière significative de ce noyau, même involontairement, et n'ont pas de définition commune du comportement raisonnable par défaut. Ainsi, quand les problèmes surgissent, la situation peut se détériorer rapidement,

les participants ne disposant pas d'un stock de réflexes culturels établis auxquels recourir pour résoudre les différends.

Cet ouvrage est un guide pratique et non une étude anthropologique ou historique. Cependant, une réelle connaissance de la culture du logiciel libre est une base essentielle pour profiter de tout conseil pratique. Une personne qui appréhende cette culture peut parcourir, en long et en large, le monde de l'Open Source, rencontrer maintes variantes locales des coutumes et des dialectes, tout en étant capable de participer partout avec aisance et efficacité. En revanche, pour qui ne la comprend pas, le processus d'organisation ou de participation à un projet sera difficile et plein de surprises. Le nombre de personnes qui développent des logiciels libres ne cessant de croître rapidement, cette dernière catégorie ne désemplit pas. C'est en grande partie une culture de nouveaux migrants, et ça continuera à l'être pendant un certain temps. Si vous pensez être l'un d'eux, la section suivante vous fournira l'arrière-plan des discussions que vous entendrez plus tard, aussi bien dans ce livre que sur Internet (d'autre part, si vous travaillez dans le monde du libre depuis un moment, vous en savez peut-être déjà pas mal sur son histoire. Dans ce cas, n'hésitez pas à sauter la prochaine section).

Historique

Le partage des logiciels est aussi ancien que les logiciels eux-mêmes. Aux premiers temps des ordinateurs, les fabricants, sentant que les bénéfices économiques se trouvaient principalement dans la production de matériel, ne prêtèrent guère attention aux logiciels et leurs atouts financiers. Un grand nombre d'acheteurs de ces premières machines étaient des scientifiques ou des techniciens capables de modifier et d'améliorer eux-mêmes les logiciels livrés avec les machines. Parfois les clients distribuaient leurs correctifs non seulement aux fabricants, mais aussi aux propriétaires de machines semblables. Les fabricants toléraient, voire encourageaient cette pratique : pour eux, l'amélioration des logiciels, peu importe la source, rendait leurs machines plus attrayantes aux yeux d'autres acheteurs potentiels.

Bien que cette période ressemblât, sous de nombreux aspects, à la culture des logiciels libres d'aujourd'hui, elle en déviait sur deux points importants. Premièrement la standardisation du matériel n'était pas vraiment à l'ordre du jour. C'était une époque d'innovation florissante pour la fabrication d'ordinateurs, mais face à la diversité des architectures, la compatibilité n'était pas une priorité. Ainsi, les logiciels écrits pour une machine ne fonctionnaient pas, en général, sur une autre. Les programmeurs avaient tendance à se spécialiser dans une architecture ou dans une famille d'architectures (alors qu'aujourd'hui, ils auraient plutôt tendance à se spécialiser dans un langage de programmation, ou une famille de langages, sachant pertinemment que leur savoir sera transférable sur n'importe quel système auquel ils se trouveraient confrontés). Comme leur expertise tendait à être spécifique à un type d'ordinateur, l'accumulation des savoirs avait pour effet de le rendre plus attractif à leurs yeux et à ceux de leurs collègues. C'était alors dans l'intérêt du constructeur de voir les codes et connaissances spécifiques à sa machine se répandre le plus possible.

Deuxièmement, Internet alors n'existait pas. Bien que les restrictions légales vis à vis du partage fussent moins fortes qu'aujourd'hui, elles étaient plus nombreuses sur le plan technique. Les moyens pour transporter les données d'un endroit à un autre étaient peu

pratiques et encombrants, comparés à aujourd'hui. Il existait des petits réseaux locaux, pratiques pour partager des informations entre employés du même laboratoire de recherche ou de la même entreprise. Mais certaines barrières restaient à surmonter si l'on voulait partager avec le monde entier tout en s'affranchissant des contraintes géographiques. Ces difficultés ont été surmontées dans de nombreux cas. Parfois des groupes entraient en contact les uns avec les autres indépendamment, en s'envoyant des disquettes ou des cassettes par courrier. Parfois les fabricants eux-mêmes centralisaient les correctifs. Un point positif était qu'une grande partie des premiers développeurs travaillaient au sein d'universités où la publication des connaissances d'une personne est attendue. Mais les réalités physiques de l'échange de données entraînaient un temps de latence, un retard proportionnel à la distance (physique ou organisationnelle) que le logiciel avait à parcourir. Le partage souple et à grande échelle, comme nous le connaissons aujourd'hui, était impossible alors.

L'avènement des logiciels propriétaires et des logiciels libres

À mesure que l'industrie mûrissait, plusieurs changements liés se sont produits. La jungle des normes matérielles a progressivement permis l'émergence de quelques lauréats grâce à une meilleure technologie, une meilleure stratégie voire la combinaison des deux. En même temps, et pas entièrement par hasard, le développement de langages de programmation dits de « haut niveau » signifiait que l'on pouvait écrire un programme une seule fois, dans un langage, et le voir automatiquement traduit (compilé) afin de fonctionner sur différents types d'ordinateurs. Les constructeurs de matériel n'ont pas manqué d'en voir les implications : un client pouvait, dès ce moment, entreprendre un travail considérable d'ingénierie logicielle sans s'enchaîner nécessairement à une architecture particulière. Ceci, couplé à une diminution des écarts de performances entre les différents ordinateurs (les conceptions les moins performantes étant évincées), faisait qu'un fabriquant qui misait tout sur le matériel pouvait s'attendre à voir baisser ses marges dans un futur proche. La puissance brute des ordinateurs n'était plus suffisante pour créer un avantage décisif, la concurrence s'est alors déplacée sur le terrain des logiciels. Vendre des logiciels, ou au moins leur donner la même importance qu'au matériel, devenait la nouvelle stratégie gagnante.

Cela signifiait, pour les fabricants, faire respecter le droit d'auteur sur leur code de manière plus stricte. Si les utilisateurs continuaient à partager et modifier le code de manière libre entre eux, ils pourraient reproduire certaines des améliorations désormais vendues en tant que « valeur ajoutée » par le fournisseur. Pire encore, le code partagé pourrait arriver entre les mains des concurrents. L'ironie est que tout ceci se passait à l'époque où Internet commençait à pointer son nez. Alors même que les obstacles tombaient, la mue du marché des ordinateurs rendit le partage de logiciels économiquement indésirable, au moins du point de vue de n'importe quelle entreprise. Les fournisseurs ont renforcé leurs positions, soit en refusant aux utilisateurs l'accès au code faisant fonctionner leurs machines, soit en imposant des accords de confidentialité rendant tout partage impossible.

Résistance consciente

Alors que le monde du libre échange de code s'affaiblissait, l'idée d'une riposte germait dans l'esprit d'au moins un programmeur : Richard Stallman. Ce dernier travaillait au laboratoire d'intelligence artificielle au MIT (Massachusetts Institute of Technology) dans les

années 1970 et au début des années 1980, durant une période qui s'est révélée être l'âge d'or du partage de code [1]. Le laboratoire avait une forte « éthique de hacker » [2] selon laquelle on n'était pas seulement encouragé à partager les améliorations apportées au système, c'était ce qui était attendu de soi. Comme Stallman écrivit plus tard :

> *« Nous n'appelions pas nos logiciels « logiciels libres » parce que ce terme n'existait pas, mais c'est bien ce qu'ils étaient. Si des gens d'une autre université ou d'une entreprise voulaient utiliser nos programmes, nous leur en donnions volontiers la permission. Si vous voyiez quelqu'un utiliser un programme intéressant que vous ne connaissiez pas, vous pouviez toujours lui en demander le code source, pour pouvoir le lire, le modifier ou en emprunter une partie pour faire un nouveau programme. »* [3]

Cette communauté utopique s'est effondrée autour de Stallman peu après 1980, quand le laboratoire a finalement été rattrapé par les changements qui s'opéraient dans le reste de l'industrie. Une start-up débaucha de nombreux programmeurs du service pour développer un système d'exploitation proche de celui sur lequel ils travaillaient, mais désormais sous licence exclusive. Au même moment, le laboratoire faisait l'acquisition de nouveaux équipements livrés avec un système d'exploitation propriétaire.

Stallman voyait très bien où cette tendance les conduisait :

> *« Les ordinateurs modernes de cette époque, comme le VAX ou le 68020 avaient leur propre système d'exploitation, mais ni l'un ni l'autre n'était libre, vous deviez signer un accord de confidentialité, même pour en obtenir une copie fonctionnelle.*
>
> *Ce qui signifiait que la première démarche à faire pour utiliser un ordinateur était de promettre de ne pas aider son voisin. Une communauté coopérative était proscrite. La règle mise en place par les créateurs des logiciels propriétaires était : 'Si vous partagez avec votre voisin vous êtes un pirate. Si vous désirez des changements, suppliez nous de les faire'. »*

Mais Stallman n'était pas un hacker comme les autres et sa personnalité le poussa à s'opposer à cette tendance. Plutôt que de continuer à travailler dans un laboratoire décimé ou de trouver un emploi de programmeur dans l'une de ces nouvelles compagnies où les fruits de son travail seraient enfermés dans une boîte, il a démissionné pour lancer le projet GNU et

1. NdT : voir à ce propos R. Stallman, S. Williams et C. Masutti, *Richard Stallman et la révolution du logiciel libre. Une biographie autorisée*, Paris, Eyrolles — Framasoft, 2010 (http ://framabook.org/stallman.html).

2. Stallman utilise le mot « hacker » pour désigner « une personne qui aime programmer et qui adore faire le malin avec ça » mais pas « quelqu'un qui s'introduit dans les ordinateurs » comme le voudrait le nouveau sens.

3. Tiré de Richard Stallman, « The Gnu project » (http ://www.gnu.org/gnu-/thegnuproject.html).

la Free Software Foundation (FSF). Le but du projet GNU [1] était de développer un système d'exploitation pour ordinateur, complètement libre et ouvert, ainsi qu'un ensemble d'applications logicielles dans lequel les utilisateurs ne seraient jamais empêchés d'hacker ou de partager leurs modifications. Il entreprenait en fait de recréer ce qui avait été détruit au laboratoire d'intelligence artificielle mais à l'échelle mondiale et sans les points faibles à l'origine de la destruction de l'esprit du laboratoire.

En plus de travailler sur le nouveau système d'exploitation, Stallman conçut une licence dont les termes garantissent que son code restera libre à tout jamais. La GNU General Public License (GPL) est une pirouette légale bien pensée : elle dit que le code peut être copié ou modifié sans restriction et que les copies ou le travail dérivé (c'est-à-dire les versions modifiées) doivent être distribuées sous la même licence que l'original, sans y ajouter de restrictions. En fait, elle utilise les lois du droit d'auteur pour produire l'effet opposé du copyright traditionnel : plutôt que de limiter la diffusion du logiciel, elle empêche quiconque, même l'auteur, de la restreindre. Pour Stallman ça valait mieux que de simplement placer son code dans le domaine public. En appartenant au domaine public, n'importe quelle copie aurait pu être incorporée dans un programme propriétaire (ce qui s'est passé avec du code publié sous des licences trop permissives). Même si cette incorporation ne diminue en rien la disponibilité continue du code originel, elle aurait signifié que les efforts de Stallman auraient pu profiter à l'ennemi : le logiciel propriétaire. La GPL peut être vue comme une forme de protection pour le logiciel libre car elle empêche les logiciels non-libres de profiter du code sous licence GPL. La GPL et ses relations avec d'autres licences de logiciels libres sont présentées en détails dans le chapitre 9.

Avec l'aide de nombreux développeurs, certains d'entre eux partageant l'idéologie de Stallman et d'autres voulant simplement voir un maximum de code libre disponible, le projet GNU commença à produire des équivalents libres pour beaucoup d'éléments importants d'un système d'exploitation. Grâce à la standardisation du matériel informatique et des logiciels, il était devenu possible d'utiliser les équivalents GNU sur des systèmes non-libres, et beaucoup l'ont fait. L'éditeur de texte GNU (Emacs) et le compilateur C (GCC) en particulier ont rencontré un succès important, rassemblant, non pour l'idéologie qu'ils véhiculaient mais simplement pour leurs mérites techniques, une grande communauté d'utilisateurs loyaux. À l'orée de 1990, GNU avait produit l'essentiel d'un système d'exploitation libre à l'exception du noyau (la partie sur laquelle démarre la machine et qui est en charge de la gestion de la mémoire, des disques et des autres ressources systèmes).

Malheureusement le projet GNU avait fait le choix d'une conception de noyau qui s'est révélée plus difficile à mettre en œuvre que prévu. Le retard qui s'ensuivit empêcha la Free Software Foundation de sortir le premier système d'exploitation libre. La dernière pièce a finalement été mise en place par Linus Torvalds, un étudiant en informatique finlandais qui, avec l'aide de volontaires du monde entier, avait compilé un noyau libre de conception plus classique. Il le nomma Linux, lequel, une fois combiné aux programmes GNU existants, donna un système d'exploitation entièrement libre. Pour la première fois vous pouviez allumer votre ordinateur et travailler sans utiliser un seul logiciel propriétaire [2].

1. GNU est l'acronyme de « GNU is Not Unix » et « GNU » dans cette extension signifie… la même chose (NdT : il s'agit d'un acronyme récursif).

2. Techniquement, Linux n'était pas le premier. Un système d'exploitation libre pour les ordinateurs compatibles IBM, appelé 386BSD, est sorti peu avant Linux. Cependant, il

La plupart des logiciels de ce nouveau système d'exploitation n'étaient pas produits par le projet GNU. En fait, GNU n'était même pas le seul groupe travaillant à l'élaboration d'un système d'exploitation libre (par exemple, le code qui deviendra NetBSD et FreeBSD était déjà en développement à cette époque). Mais l'importance de la Free Software Foundation n'était pas seulement dans le code écrit par ses membres : il s'agissait aussi d'un discours idéologique. En mettant en avant le logiciel libre comme une cause à part entière, plutôt qu'une commodité, ils lui ont donné une dimension politique difficile à ignorer par les programmeurs. Même ceux en désaccord avec la FSF ont dû prendre en compte cette question, parfois pour revendiquer une position différente. L'efficacité du message de la FSF tient au fait qu'il est lié au code, grâce à la GPL et d'autres textes. Alors que le code se répand partout, le message se diffuse également.

Résistance accidentelle

Bien que le champ naissant du logiciel libre ait été très dynamique, peu d'activités furent aussi clairement idéologiques que le projet GNU de Stallman. L'un des plus importants était le projet Berkeley Software Distribution (BSD), une ré-implémentation graduelle du système d'exploitation Unix (qui jusqu'à la fin des années 1970 fut un projet de recherche vaguement propriétaire chez AT&T) par des programmeurs de l'Université de Californie de Berkeley. Le groupe BSD ne prit pas ouvertement position sur la nécessité des programmeurs de s'unir et de partager, mais ils mirent en pratique cette idée avec style et enthousiasme en coordonnant un gros effort de développement coopératif grâce auquel les lignes de commande des utilitaires Unix, les librairies de code et finalement le noyau du système d'exploitation lui-même furent ré-écrits entièrement, principalement par des volontaires. Le projet BSD devint un exemple important de développement non-idéologique de logiciel libre et servit également de terrain d'entraînement à de nombreux développeurs qui continueront par la suite à être actifs dans le monde de l'Open Source.

Un autre nid de développement coopératif fut le système *X Window*, un environnement graphique, libre et transparent au réseau, développé au MIT au milieu des années 1980 en partenariat avec des vendeurs de matériel ayant un intérêt commun à pouvoir proposer à leurs clients un système graphique. Loin de s'opposer aux logiciels propriétaires, la licence X permettait délibérément l'ajout d'extensions propriétaires au cœur libre du système, chaque membre du consortium souhaitant pouvoir améliorer la distribution X de base et, par ce moyen, obtenir un avantage concurrentiel sur les autres. X Window [1] en lui même était un logiciel libre, mais essentiellement par volonté de placer les concurrents sur un même pied d'égalité, et non pas comme un désir quelconque de mettre un terme à l'hégémonie des logiciels propriétaires. Devançant le projet GNU de quelques années, TeX, le système libre de traitement de texte de Donald Knuth permettant la création de documents de grande qualité, en est un autre exemple. Il fut publié sous une licence permettant à quiconque de modifier et de distribuer le code mais qui interdisait de nommer le résultat « TeX » s'il n'avait pas passé

était bien plus compliqué de faire marcher 386BSD. Linux a créé des remous non seulement parce qu'il était libre mais aussi parce qu'il avait vraiment une bonne chance de faire marcher l'ordinateur sur lequel vous l'aviez installé.

1. Ils préfèrent qu'il soit appelé le « Système X Window », mais en pratique on l'appelle « X Window » parce que les trois mots sont trop encombrants.

une série de tests de compatibilité stricts (c'est là un exemple de licence libre de « protection de marque déposée » que nous approfondirons dans le chapitre 9). Knuth n'exprimait aucun avis, d'une façon ou d'une autre, sur la question de l'opposition des logiciels libres aux logiciels propriétaires : il cherchait simplement un meilleur traitement de texte pour achever son véritable but, un livre sur la programmation informatique, et ne voyait aucune raison de ne pas offrir son système au monde une fois celui-ci prêt.

Sans faire une liste de tous les projets et de toutes les licences, on peut dire qu'à la fin des années 1980, il existait de nombreux logiciels disponibles sous une grande variété de licences. La diversité des licences reflétait une diversité équivalente des motivations. Même certains des programmeurs choisissant la GNU GPL n'étaient pas aussi déterminés idéologiquement que le projet GNU lui-même. Et s'ils appréciaient de travailler sur des logiciels libres, nombre d'entre eux ne considéraient pas les logiciels propriétaires comme un mal social. Quelques uns ressentaient un besoin moral de débarrasser le monde des « logiciels panneaux d'affichage » (appellation de Stallman pour les logiciels propriétaires), mais d'autres étaient plutôt motivés par l'émulation technique ou le plaisir de travailler avec des collaborateurs partageant leurs idées, voire par le simple désir humain de gloire. Pourtant, ces diverses motivations n'interagissaient pas généralement de manière destructive. C'est en partie dû au fait que les logiciels, contrairement à d'autres œuvres créatives comme la prose ou les arts visuels, doivent réussir des tests semi-objectifs pour être considérés comme des succès : ils doivent fonctionner et ne pas comporter trop de bogues. Cela donne aux participants du projet une base commune, une raison et un cadre pour travailler ensemble sans se préoccuper des qualifications autres que techniques.

Les développeurs avaient une autre raison de rester unis : il s'est avéré que le code produit par le monde des logiciels libres était de très bonne qualité. Dans certains cas, ces programmes étaient d'un point de vue technique nettement supérieurs à leurs équivalents non-libres, dans d'autres cas ils étaient au moins comparables, et bien sûr, toujours meilleur marché. Alors que seulement quelques personnes auraient pu être motivées pour produire des logiciels libres sur des bases strictement philosophiques, beaucoup l'étaient du fait des meilleurs résultats obtenus. De plus, il y avait toujours un certain pourcentage d'utilisateurs prêts à donner de leur temps et de leurs compétences afin d'aider à maintenir et améliorer le logiciel.

Cette tendance à produire du bon code n'était certainement pas universelle, mais cela se renouvelait à une fréquence croissante dans les projets de logiciels libres du monde entier. Les entreprises tributaires de logiciels commencèrent progressivement à le remarquer. Beaucoup d'entre elles découvrirent qu'elles utilisaient déjà, sans le savoir, des logiciels libres pour leurs opérations quotidiennes (les dirigeants ne sont pas toujours au courant des choix de leur service informatique). Les sociétés entreprirent de s'impliquer davantage dans les projets de logiciels libres en mettant à disposition du temps et des équipements, voire même en finançant le développement. De tels investissements pouvaient, dans les meilleurs scénarios, se montrer très lucratifs par un coefficient de retour conséquent. Le commanditaire ne paie qu'un nombre réduit de programmeurs experts pour se consacrer à plein temps au projet, mais récolte les fruits de la collaboration de chacun, y compris du travail des bénévoles et des développeurs payés par d'autres sociétés.

« Libre » contre « Open Source »

Alors que le monde de l'entreprise prêtait de plus en plus attention aux logiciels libres, les programmeurs se trouvèrent confrontés à de nouveaux problèmes de représentation. L'un d'entre eux était le mot « free » lui-même. Entendant pour la première fois « free software », nombre de personnes pensaient, à tort, que cela signifiait « logiciel gratuit ». S'il est vrai que tous les logiciels libres ne coûtent rien [1], tous les logiciels gratuits ne sont pas libres. Par exemple, durant la bataille des navigateurs dans les années 90 et dans la course aux parts de marché qu'ils se livraient, Netscape et Microsoft offraient leurs navigateurs Web gratuitement. Aucun des deux n'était libre dans le sens des « logiciels libres ». Vous ne pouviez pas obtenir le code source et, même si vous y parveniez, vous n'aviez pas le droit de le modifier ni de le redistribuer [2]. Vous pouviez tout juste télécharger un exécutable et le lancer. Les navigateurs n'étaient pas plus libres que les logiciels sous film plastique achetés en magasin : tout au plus étaient-ils diffusés à prix inférieur.

La confusion sur le mot « free » est entièrement due à l'ambivalence malheureuse du terme en anglais. La plupart des autres langues font une distinction entre la notion de prix et de liberté (la différence entre gratuit et libre est évidente dans une langue romane par exemple). Mais l'anglais étant *de facto* la langue d'échange sur Internet, le problème spécifique à cette langue concerne, à un certain degré, tout le monde. L'incompréhension liée au mot *free* était telle que les programmeurs de logiciels libres ont fini par créer une formule en réponse : « C'est *free* (libre) comme dans *freedom* (liberté), pensez à *free speech* (liberté de parole), pas à *free beer* (bière gratuite) ». Reste que devoir l'expliquer sans cesse est fatigant. De nombreux programmeurs ressentaient, à juste titre, que l'ambiguïté du mot *free* rendait plus difficile la compréhension par le public de ce type de logiciels.

Cependant, le problème apparaissait plus profond que cela. Le mot « libre » était vecteur d'une connotation morale indéniable : si la liberté était une fin en soi, peu importait que les logiciels libres soient également meilleurs ou plus profitables à certaines sociétés dans certaines circonstances. Ce n'étaient là que les effets secondaires bienvenus d'une motivation qui, au fond, n'était ni technique ni commerciale mais morale. En outre, l'idée de « Libre comme dans liberté » imposait une flagrante contradiction aux sociétés souhaitant soutenir certains logiciels libres dans un domaine particulier de leurs activités, mais voulant continuer à commercialiser des logiciels propriétaires dans d'autres branches.

Ces dilemmes touchèrent une communauté déjà promise à une crise d'identité. Les programmeurs qui écrivent des logiciels libres n'ont jamais réussi à se mettre d'accord sur le but final, s'il existe, du mouvement des logiciels libres. Même dire que les opinions vont d'un extrême à l'autre serait trompeur, car cela implique une vision linéaire au lieu de la dispersion multidimensionnelle existante. Cependant, on peut définir deux grands courants de pensée, si vous acceptez de passer outre les détails pour le moment. L'un des groupes partage la pensée de Stallman que la liberté de partager et modifier est la plus importante, et

1. On peut faire payer quelque chose en échange de copies d'un logiciel libre, mais comme on ne peut pas empêcher l'acheteur de le redistribuer gratuitement, le prix tend immédiatement vers zéro.

2. Finalement, le code source de Netscape Navigator a été publié sous une licence libre, en 1998, et devint la base du navigateur Web Firefox. Voir http ://www.mozilla.org.

qu'en conséquence, si vous cessez de parler de liberté, vous abandonnez l'idée principale. Pour d'autres, c'est le logiciel qui compte et ils ne se voient pas dire que les logiciels propriétaires sont par définition mauvais. Certains programmeurs de logiciels libres, mais pas tous, pensent que l'auteur (ou l'employeur dans le cas de travail rémunéré) devrait avoir le droit de contrôler les termes de la distribution et qu'aucun jugement moral ne devrait été rattaché au choix de termes particuliers.

Pendant longtemps personne n'a vraiment eu à prêter attention à ces différences ou à leurs interférences, mais le succès grandissant des logiciels libres dans le monde de l'entreprise a rendu cette question inévitable. En 1998, le terme « Open Source » a été inventé, en tant qu'alternative à « libre », par une coalition de programmeurs devenue par la suite The Open Source Initiative (OSI [1]). Pour l'OSI non seulement le terme « logiciel libre » était déroutant, mais le mot « libre » n'était que le symptôme d'un problème plus général : le mouvement avait besoin d'une stratégie de commercialisation pour s'adapter au monde de l'entreprise, et les discours sur les avantages moraux et sociaux du partage ne pénètreraient pas les conseils d'administration des entreprises. Selon eux :

> *« L'Open Source Initiative est un programme de commercialisation des logiciels libres. C'est un argumentaire en faveur des 'logiciels libres' qui s'appuie sur une solide base pragmatique plutôt que sur une idéologie dévote. La substance gagnante n'a pas changé, l'attitude et le symbolisme perdants, eux, ont changés.*
>
> *Le point qui doit être exposé à la plupart des techniciens n'est pas le concept de l'Open Source, mais le nom. Pourquoi ne pas l'appeler, comme nous l'avons toujours fait, logiciel libre ?*
>
> *Une raison simple est que le terme 'logiciel libre' peut facilement prêter à confusion et mener au conflit...*
>
> *Mais la vraie motivation de ce changement de nom est économique. Nous essayons maintenant de vendre notre concept au monde de l'entreprise. Nous avons un produit compétitif, mais notre positionnement, dans le passé, était très mauvais. Le terme 'logiciel libre' n'a pas été compris correctement par les hommes d'affaires qui ont pris le désir de partage pour de l'anti-capitalisme, ou pire encore, pour du vol.*
>
> *Les décideurs des principales grandes entreprises n'achèteront jamais un 'logiciel libre'. Mais si nous prenons les mêmes idées, les mêmes licences de logiciel libre et que l'on remplace le nom par 'Open Source' ? Alors ils achèteront. Certains hackers ont du mal à y croire, mais c'est parce que ce sont des techniciens qui pensent aux choses concrètes, palpables et qui ne comprennent pas l'importance de l'image quand vous vendez quelque chose.*
>
> *Dans le monde des affaires, l'apparence est une réalité. L'apparence que nous voulons abattre les barricades et travailler avec le monde des*

1. La page Web de l'OSI est : http ://www.opensource.org.

> *affaires compte au moins autant que la réalité de notre comportement, nos convictions et nos logiciels. »* [1]

La pointe de l'iceberg de la polémique apparaît dans ce texte. Il mentionne « nos convictions » mais évite intelligemment de citer précisément quelles sont ces convictions. Pour certains, ça peut être la conviction que le code développé selon un processus ouvert sera meilleur, pour d'autres ça peut-être la conviction que toutes les informations devraient être partagées. L'usage du mot « vol » fait (sûrement) référence à la copie illégale, dont beaucoup se défendent : ce n'est pas un vol si personne n'est dépossédé de son bien. On retrouve aussi le raccourci facile, mais injuste, entre logiciels libres et anti-capitalisme, sans débattre du bien fondé de cette accusation.

Rien de tout cela ne signifie que le site de l'OSI est incomplet ou trompeur. Il ne l'est pas. Au contraire, c'est la vitrine de ce qui manquait au mouvement du logiciel libre d'après l'OSI : une bonne stratégie commerciale où « bon » veut dire : « viable dans le monde des affaires ». L'Open Source Initiative a offert à beaucoup de gens exactement ce qu'ils cherchaient : un vocabulaire pour parler des logiciels libres avec un plan de développement et une stratégie commerciale, plutôt qu'une croisade idéologique.

L'apparition de l'Open Source Initiative a transformé l'horizon des logiciels libres. Elle a reconnu une dichotomie longtemps inavouée et, par là même, a forcé le mouvement à reconnaître qu'il avait autant une politique interne qu'externe. On voit aujourd'hui que les deux groupes ont dû trouver un terrain d'entente puisque la plupart des projets font participer des programmeurs des deux camps aussi bien que d'autres n'entrant pas clairement dans l'une ou l'autre des catégories. Cela ne veut pas dire que les gens ne parlent jamais de motivations idéologiques ; on fait parfois référence aux manquements à la « morale de hacker » traditionnelle par exemple. Mais il est rare de voir un développeur de l'un des deux mondes douter ouvertement des motivations profondes de ses collègues. La contribution transcende le participant. Si quelqu'un produit du bon code, personne ne lui demande s'il le fait pour des raisons morales, ou parce que son employeur le paie pour cela, ou parce qu'il étoffe son Curriculum Vitae ou quoi que ce soit d'autre. L'évaluation et les critiques de la contribution se font sur des critères techniques. Même des organisations ouvertement politiques comme le projet Debian dont le but est d'offrir un environnement 100% libre (c'est-à-dire « libre comme dans liberté ») sont plutôt ouvertes à l'intégration de code non-libre et à la coopération avec des programmeurs qui ne partagent pas exactement les mêmes buts.

La situation actuelle

Quand vous dirigez un projet de logiciel libre, vous n'avez pas besoin de discuter de lourds problèmes philosophiques tous les jours. Les programmeurs n'ont pas à cœur de convertir les autres participants à leurs opinions diverses (ceux qui le font se retrouvent rapidement incapables de travailler sur un projet). Mais vous devez savoir que la question « libre » contre « Open Source » existe, au moins pour éviter de dire des choses inamicales à certains

1. D'après le site http ://www.opensource.org/. Ou plutôt une de ses versions anciennes — l'OSI a apparemment ôté ces pages depuis lors, bien qu'elles soient toujours visibles sur http ://web.archive.org/.

des participants, mais aussi parce que comprendre les motivations des développeurs est la meilleure manière, la seule manière en un certain sens, de diriger un projet.

Le logiciel libre est une culture par choix. Pour y naviguer avec succès, vous devez comprendre pourquoi les gens ont fait le choix d'y participer en premier lieu. Les manières coercitives ne fonctionnent pas. Si les gens ne sont pas heureux au cœur d'un projet, ils vont simplement s'en écarter pour en rejoindre un autre. Le logiciel libre est remarquable, même au sein des communautés de volontaires, par la légèreté de l'investissement. La plupart des gens engagés n'ont jamais vraiment rencontrés d'autres participants face à face, et donnent simplement un peu de leur temps quand ils en ressentent l'envie. Les moyens usuels, par lesquels les humains établissent des liens entre eux et forment des groupes qui durent, sont restreints à leur plus simple expression : des mots écrits transmis par des fils électriques. À cause de cela, la formation d'un groupe soudé et dévoué peut prendre du temps. Inversement, un projet peut facilement perdre un volontaire potentiel dans les cinq premières minutes de présentation. Si un projet ne fait pas bonne impression, les nouveaux venus lui donnent rarement une seconde chance.

La brièveté, ou plutôt la brièveté potentielle, des relations est peut-être l'obstacle le plus intimidant au commencement d'un nouveau projet. Qu'est ce qui persuadera tous ces gens de rester groupés suffisamment longtemps pour produire quelque chose d'utile ? La réponse à cette question est suffisamment complexe pour être le sujet du reste de ce livre, mais si elle devait être exprimée en une seule phrase, ce serait :

> *« Les gens doivent sentir que leur relation à un projet, et leur influence sur celui-ci, est directement proportionnelle à leurs contributions. »*

Aucune catégorie de développeurs, ou de développeurs potentiels, ne devrait se sentir mise à l'écart ou être discriminée pour des raisons non-techniques. Les projets portés par une société et/ou des développeurs salariés doivent y faire particulièrement attention, le chapitre 5 aborde ce sujet plus en détail. Bien sûr, cela ne veut pas dire que, si vous ne bénéficiez pas du financement d'une société, vous n'avez à vous soucier de rien. L'argent n'est que l'un des nombreux facteurs pouvant influencer le succès d'un projet. Il y a aussi les questions de choix du langage, de la licence, du processus de développement, du type précis d'infrastructure à mettre en place, de la manière efficace de rendre publique la base du projet et bien plus encore. Commencer un projet du bon pied est le sujet du prochain chapitre.

Bien démarrer

Eric Raymond, dans un article maintenant célèbre sur les processus Open Source intitulé « La Cathédrale et le Bazaar [1] » décrit les étapes classiques du lancement d'un projet de logiciel libre. Il y écrit :

> *« Tout bon logiciel commence par gratter un développeur là où ça le démange. »*

Remarquez que Raymond ne dit pas que les projets Open Source démarrent seulement quand quelques individus ont une démangeaison. Il dit plutôt que tout *bon* logiciel est d'abord la réponse qu'apporte un programmeur à un problème qui l'irrite personnellement. C'est exactement ce qui se passe dans le monde du logiciel libre : les besoins personnels sont les motivations les plus fréquentes pour commencer un projet. C'est aujourd'hui encore la motivation de base de la plupart des projets libres, mais moins qu'en 1997 quand Raymond écrivait ces lignes. Aujourd'hui nous voyons des entreprises, y compris à but tout à fait lucratif, qui lancent de vastes projets Open Source à gestion centralisée en partant de zéro. Le programmeur isolé, écrivant des bouts de code pour résoudre un problème local précis et qui se rend compte que le résultat obtenu peut être largement étendu, est encore à la base de nombreux nouveaux logiciels libres, mais d'autres modèles ont émergé.

L'idée de Raymond demeure très pertinente. Il faut que les créateurs du programme aient un réel intérêt à le voir aboutir car ils l'utilisent eux-mêmes : c'est une condition essentielle. Si le logiciel ne fait pas ce qu'il est censé faire, la personne ou l'organisation commanditaire ne seront pas satisfaites dans leur travail au quotidien. Prenons l'exemple du projet OpenAdapter [2], lancé par la banque d'investissement Dresdner Kleinwort Wasserstein. Il s'agit

1. http ://catb.org/esr/writings/homesteading/
2. https ://www.openadapter.org/

d'une infrastructure Open Source pour l'intégration de systèmes d'informations financières disparates : on peut difficilement dire que ce projet réponde au besoin particulier d'un quelconque développeur mais plutôt à celui d'une institution. Mais c'est bien la réponse concrète à une insatisfaction de l'institution et de ses partenaires. En conséquence, si le programme faillit à sa mission, ils le sauront. Cette configuration produit de bons logiciels parce que la boucle de rétroaction est efficace. Le programme n'est pas écrit pour être vendu à un client quelconque afin qu'il puisse résoudre *son* problème. C'est nous qui l'écrivons pour résoudre *notre propre problème* et pour partager ensuite la solution avec tous, un peu comme si le problème était une maladie et le logiciel le médicament dont la diffusion pourrait éradiquer l'épidémie.

Ce chapitre traite de l'élaboration et de la publication d'un nouveau projet de logiciel libre, mais de nombreuses recommandations sembleraient familières aux organisations médicales qui diffusent des médicaments. Les buts paraissent très similaires : vous voulez expliquer de manière simple ce que le médicament fait, le mettre entre les mains des personnes concernées et vous assurer que ceux qui le reçoivent savent bien l'utiliser ; dans le cas des logiciels, vous voulez aussi inciter certains destinataires à rejoindre la recherche en cours pour améliorer le médicament.

La distribution du logiciel libre comporte deux facettes : elle doit trouver des utilisateurs d'une part et des développeurs de l'autre. Ces deux besoins ne sont pas forcément antagonistes même s'ils rendent la présentation initiale du projet plus compliquée. Certaines informations sont utiles aux deux publics, d'autres le sont uniquement à l'un ou à l'autre. Les deux types d'informations devraient répondre au principe de présentation adaptée, ce qui signifie que le degré de détail présenté à chacun devrait correspondre directement à la somme de temps et d'efforts consentis par le lecteur. Un effort plus important devrait toujours être récompensé à sa juste valeur. Quand le retour sur investissement n'est pas satisfaisant, les gens peuvent rapidement perdre la foi et arrêter de s'investir.

Le corollaire en est que *les apparences comptent vraiment*. Les programmeurs, en particulier, sont souvent sceptiques à ce sujet. Leur amour pour le fond plutôt que pour la forme est presque une source de fierté professionnelle. Ce n'est pas un hasard si tant de programmeurs montrent de l'antipathie envers le marketing et les relations publiques, ou si les graphistes de métier sont souvent horrifiés par ce que les développeurs peuvent produire.

C'est dommage car il existe des situations — la présentation d'un projet en est une — où la forme fait partie du tout. Par exemple, l'une des premières choses que retient un visiteur à propos d'un projet est l'aspect de son site Web. Cette information est absorbée avant que le vrai contenu du site ne soit compris, avant que le texte ne soit lu ou que le visiteur ne clique sur les liens. Même si cela peut être injuste, les gens ne peuvent s'empêcher de se forger une première impression. L'apparence du site montre l'application apportée à la présentation du projet. Les humains ont des antennes très sensibles pour détecter le souci du détail. La plupart d'entre nous peuvent dire en un coup d'œil si un site Web a été assemblé à la va-vite ou s'il a été mûrement réfléchi. C'est la première information que votre projet montre et l'impression qu'elle crée pèsera, par association, sur le reste du projet.

En conséquence, alors que ce chapitre du livre porte sur le contenu à la base de votre projet, souvenez-vous que son apparence compte également. Étant donné que le site Web s'adresse à deux types de visiteurs, les utilisateurs et les développeurs, une attention particulière doit être apportée à sa clarté et sa concision. Bien que ça ne soit pas le meilleur endroit

pour traiter de la conception d'une page Web, un principe est suffisamment important pour être mentionné ici, particulièrement quand le site s'adresse à plusieurs publics (qui peuvent se confondre) : les gens devraient avoir une vague idée de la direction d'un lien avant de cliquer dessus. Par exemple, il devrait être évident que les liens vers la documentation utilisateur ne renvoient pas à la documentation développeur. Diriger un projet consiste à la fois à fournir des informations mais aussi à rassurer. La simple présence de certaines offres standard, à l'endroit attendu, rassure les utilisateurs et les développeurs qui décident s'ils vont ou non s'impliquer. Cela montre que le projet a une ligne de conduite définie, qu'il a anticipé les questions des visiteurs, et qu'il a fait l'effort d'y répondre de manière simple. Quand le projet témoigne d'une préparation soignée, c'est comme s'il envoyait un message : « Vous n'allez pas perdre votre temps en nous rejoignant », ce qui est exactement ce que les gens veulent entendre.

Regardez d'abord autour de vous

Avant de commencer un projet Open Source, faites bien attention :

Regardez toujours autour de vous afin de savoir s'il n'existe pas déjà un projet correspondant à ce que vous souhaitez. Il y a de bonnes chances que, quel que soit le problème à résoudre, quelqu'un ait déjà pris l'initiative avant vous. Si le problème a été résolu et son code rendu public, sous une licence libre, vous n'avez aucune raison de réinventer la roue. Il y a bien sûr des exceptions : si vous voulez débuter un projet à des fins d'expérience pédagogique, un code pré-existant ne vous aidera pas, ou encore : le projet que vous avez en tête est si spécialisé que vous savez qu'il y a peu de chance qu'un autre l'ait déjà fait. Il n'y a, en général, aucune raison de ne pas effectuer de recherche, le gain peut être énorme. Si les moteurs de recherche sur Internet ne retournent aucune réponse, tentez votre chance sur freshmeat.net [1] (un site rassemblant les nouvelles à propos de projets Open Source dont je parlerai plus longuement par la suite), sur sourceforge.net [2] et dans le répertoire des logiciels libres [3] de la Free Software Foundation. Même si vous ne trouvez pas exactement votre bonheur, vous pourriez découvrir un projet s'en approchant suffisamment pour qu'il soit plus logique d'y ajouter une fonctionnalité en le rejoignant que de commencer à partir de rien et par vous-même.

1. Commencez avec ce que vous avez

Vous avez cherché mais n'avez rien trouvé correspondant à votre besoin, vous êtes donc décidé à commencer un nouveau projet.

Que faire maintenant ?

La partie la plus difficile du lancement d'un projet de logiciel libre est la transposition d'une vision personnelle en une vision publique. Vous, ou votre organisation, savez parfaitement ce que vous voulez, mais expliquer ce but de manière compréhensible au monde

1. http ://freshmeat.net/
2. http ://www.sourceforge.net/
3. http ://directory.fsf.org/

entier représente un travail important. Il est essentiel cependant de prendre le temps de le faire. Vous et les autres membres fondateurs devez décider d'une définition précise du projet (c'est-à-dire, décider de ses limites, de ce qu'il ne réalisera pas aussi bien que de ce qu'il fera) et écrire une déclaration d'objectif (ou mission). Cette partie n'est, en général, pas trop compliquée bien qu'elle puisse révéler des non-dits et même des désaccords à propos de la nature du projet, ce qui est une bonne chose : mieux vaut résoudre ces problèmes maintenant que plus tard. L'étape suivante est de préparer la présentation du projet au public ; une véritable corvée.

Ce qui rend cette étape si laborieuse est qu'il s'agit ici principalement d'organiser et de documenter des choses que tout le monde connaît déjà, « tout le monde » signifiant ici ceux qui ont déjà été impliqués dans un projet. Par conséquent, pour les gens s'occupant de cette tâche, il n'y a pas de gain immédiat. Ils n'ont pas besoin de fichier « Lisez moi » décrivant le projet, ni des fiches techniques ou de la documentation utilisateur. Ils n'ont pas besoin d'une arborescence du code établie avec attention selon des standards officieux mais largement utilisés pour la distribution du code de logiciels. Peu importe comment le code source est organisé, ils s'y retrouveront, ils y sont déjà habitués, et si le code fonctionne, ils savent l'utiliser. Peu importe aussi pour eux si les bases architecturales du projet restent non documentées, ils en sont déjà familiers.

Les nouveaux arrivants, d'un autre côté, ont besoin de ces informations. Heureusement, ils n'ont pas besoin de tout, tout de suite. Il n'est pas nécessaire de donner accès à toutes les sources possibles avant de rendre le projet public. Dans un monde parfait, peut-être, chaque nouveau projet Open Source commencerait avec un modèle de document exhaustif, un manuel utilisateur complet (avec des avertissements pour les fonctionnalités prévues mais pas encore implémentées), un beau code bien assemblé et portable, capable de fonctionner sur n'importe quelle plateforme, etc. En réalité, s'occuper de tous ces détails prendrait un temps considérable et serait rédhibitoire : de toute façon, c'est un travail pour lequel on peut espérer recevoir de l'aide des volontaires une fois le projet sur les rails.

Il est nécessaire cependant de s'investir suffisamment dans la présentation afin que les débutants puissent surmonter les premiers obstacles dus à leur manque d'habitude. Il faut le voir comme le premier pas pour lancer le projet du bon pied en lui apportant une sorte d'énergie d'activation minimale. Certains appellent ce seuil l'*énergie d'hacktivation* : l'énergie qu'un débutant doit fournir avant de recevoir quelque chose en retour. Plus le niveau requis d'énergie d'hacktivation est bas, mieux ça vaut. Votre première tâche sera d'abaisser cette énergie d'hacktivation à un niveau encourageant les gens à s'engager.

Chacune des sections suivantes décrit un aspect important du commencement d'un nouveau projet. Elles sont plus ou moins présentées dans l'ordre d'apparition pour un nouveau visiteur, même si, bien sûr, votre ordonnancement peut être différent. Vous pouvez vous en servir comme d'une liste de contrôle. Quand vous commencez un projet, suivez cette liste et assurez-vous que chaque point est traité, ou au moins que vous n'aurez pas de problèmes quant aux conséquences possibles si vous décidez d'en écarter un.

Choisissez un bon nom

Mettez-vous à la place de quelqu'un venant de découvrir votre projet, peut-être en le trouvant par hasard lors d'une recherche de logiciel. La première chose qu'il verra est le nom du projet.

Un bon nom ne fera pas automatiquement de votre projet un succès et un mauvais ne le condamnera pas (enfin, un très mauvais nom peut le faire, mais nous partons de l'hypothèse que personne ici ne tente intentionnellement de faire échouer son projet). Quoi qu'il en soit, un mauvais nom peut ralentir l'adoption du projet parce que les gens ne le prennent pas au sérieux, ou simplement parce qu'ils ont du mal à le retenir.

Un bon nom doit avoir les caractéristiques suivantes :

– Il donne une idée de l'objectif du projet, ou présente au moins clairement son domaine : ainsi, celui qui connaît le nom et la teneur du projet l'aura plus facilement à l'esprit par la suite.
– Il est simple à mémoriser. Ici, on ne peut ignorer que l'anglais est devenu le langage par défaut sur Internet : « simple à mémoriser » signifie « simple à mémoriser pour quelqu'un parlant anglais ». Un jeu de mots dépendant de la prononciation du locuteur dans cette langue, par exemple, sera obscur pour les nombreux lecteurs dont l'anglais n'est pas la langue maternelle. Si le jeu de mot est particulièrement bien trouvé et facile à mémoriser, cela peut tout de même fonctionner : gardez simplement à l'esprit qu'en voyant le nom, beaucoup ne l'entendront pas mentalement comme une personne de langue maternelle anglaise.
– Il n'est pas le même que celui d'un autre projet et ne s'approche d'aucune marque déposée. Ce ne sont que des bonnes manières et du bon sens légal. Vous ne voulez pas créer une confusion d'identité. Il est déjà assez difficile de suivre tout ce qui est disponible sur le Net sans s'encombrer de problèmes d'homonymie. Les sites mentionnés précédemment dans l'introduction vous seront d'une grande aide pour savoir si un projet a déjà adopté le nom auquel vous pensiez. Une recherche gratuite de marques déposées est également possible sur les sites nameprotect.org [1] et uspto.org [2].
– il est disponible comme nom de domaine en .com, .net ou .org si possible. Vous devriez en choisir un, vraisemblablement en .org, comme site officiel de présentation du projet : les deux autres devraient renvoyer à ce premier site et ne sont là que pour éviter une confusion d'identité autour du nom du projet. Même si vous avez l'intention de l'héberger sur un autre site (voir la section « Forges »), vous pouvez toujours enregistrer des noms de domaines particuliers pour le projet et les faire renvoyer vers le site d'hébergement. Cela aide beaucoup les utilisateurs d'avoir une adresse simple et facilement mémorisable.

Définissez clairement vos objectifs

Une fois le site du projet trouvé, les gens chercheront une description rapide de sa teneur, votre déclaration d'intention, afin de pouvoir décider rapidement (dans les 30 secondes) s'ils

1. http ://www.nameprotect.org/
2. http ://www.uspto.gov/

souhaitent en savoir plus ou pas. Cette description devrait avoir une place de choix sur la première page, de péférence juste en-dessous du nom du projet.

Votre déclaration doit être concrète, restreinte et surtout concise. En voici un bon exemple tiré du site openoffice.org [1] :

> *« Pour créer, en tant que communauté, la suite bureautique internationale dominante qui fonctionnera sur toutes les plateformes principales et qui donnera accès à toutes les fonctionnalités et à toutes les données au travers d'API basées sur des composants libres et des formats de fichier basés sur XML. »*

En seulement quelques mots, tous les points importants sont traités, principalement en s'appuyant sur les connaissances antérieures du lecteur. En écrivant « en tant que communauté », ils annoncent qu'aucune société ne dominera le développement, « internationale » veut dire que les gens pourront utiliser le logiciel dans de nombreuses langues et dialectes, « toutes les plateformes principales » veut dire qu'il sera utilisable sous Unix, Macintosh et Windows. La suite annonce que l'utilisation d'interfaces ouvertes et de formats de fichiers compréhensibles est une part importante de l'objectif visé. Ils ne disent pas de but en blanc qu'ils essaient de proposer une alternative libre à Microsoft Office, cependant la plupart des gens peuvent le lire entre les lignes. Bien que cette déclaration d'intention puisse paraître large au premier abord, elle est assez concise : les mots « suite bureautique » signifient quelque chose de très concret aux familiers de ce genre de logiciels. Encore une fois, la présumée connaissance préalable du lecteur (dans ce cas probablement de MS Office) est utilisée pour que la déclaration reste concise.

La nature de l'objectif énoncé dépend en partie de son auteur, pas uniquement du logiciel décrit. Par exemple, il est logique pour OpenOffice.org d'utiliser les mots « en tant que communauté » car le projet a été initié et est encore largement financé par Sun Microsystems. En ajoutant ces mots, Sun comprend et rassure ceux qui pourraient craindre sa domination sur le processus de développement. En montrant simplement que l'on a conscience des conséquences possibles d'un problème, un grand pas en avant est fait pour l'éviter complètement. D'un autre côté, les projets dont le soutien financier n'est pas assuré par une unique entreprise n'ont pas besoin de tenir un tel discours ; après tout, le développement communautaire est la norme : il devrait donc être normalement inutile de le lister parmi les objectifs.

Indiquez que le projet est libre

Ceux qui sont toujours intéressés, après avoir lu la déclaration d'intention, vont ensuite vouloir plus de détails, peut-être une documentation utilisateur ou développeur, et finiront par télécharger quelque chose. Mais auparavant, ils voudront s'assurer que c'est bien Open Source.

La première page doit annoncer sans ambiguïté que le projet est Open Source. Cela peut paraître évident, mais vous seriez surpris de voir combien de projets oublient de le faire. J'ai vu des sites de logiciels libres où la première page, non seulement ne donnait pas la

1. http ://www.openoffice.org/

licence de distribution du logiciel, mais ne déclarait même pas non plus que le logiciel était libre. Parfois l'information cruciale était reléguée à la page Téléchargements ou à la page Développeurs, voire une autre encore, nécessitant plus d'un clic pour s'y rendre. Dans les cas extrêmes, la licence n'était pas du tout mentionnée sur le site, le seul moyen de la découvrir étant de télécharger le logiciel pour trouver cette information.

Ne commettez pas cette erreur. Une telle omission peut vous faire perdre de nombreux développeurs et utilisateurs potentiels. Annoncez de manière claire, juste en-dessous de la mission, que le projet est un « logiciel libre » ou un « logiciel Open Source » et donnez la licence exacte. Un guide rapide pour choisir la licence est proposé dans la section nommée « Choisir une licence et l'appliquer » plus loin dans ce chapitre, et les problèmes relatifs aux licences sont exposés plus en détails dans le chapitre 9.

À ce stade, notre visiteur a établi (probablement en une minute ou moins) s'il compte passer, disons, au moins cinq minutes de plus à s'informer sur le projet. Voyons maintenant ce qu'il devrait trouver durant ces cinq minutes.

Liste des fonctionnalités et pré-requis

Il devrait y avoir une brève liste des fonctionnalités proposées par le logiciel (si un aspect n'est pas encore finalisé, vous pouvez toujours le lister en ajoutant à côté « prévu » ou « en cours ») et l'environnement informatique requis pour l'utiliser. Pensez la liste fonctionnalités / pré-requis comme si vous vous adressiez à quelqu'un vous demandant un rapide résumé du logiciel. C'est une extension logique de la mission. Ainsi, cela pourrait être :

> *« Créer un outil d'indexation de textes entiers et un moteur de recherche au moyen d'une API riche, en vue de son utilisation par des programmeurs fournissant des services de recherche concernant de grandes collections de fichiers textes. »*

La liste des fonctionnalités et des pré-requis devrait donner des détails en clarifiant la portée de la mission.

Fonctionnalités :

- recherche de texte brut, HTML et XML,
- recherche de mots ou de phrases,
- (prévu) correspondance approximative,
- (prévu) mise à jour incrémentale de l'index,
- (prévu) indexation de sites Web distants.

Pré-requis :

- python 2.2 ou plus récent,
- suffisamment d'espace disque pour contenir l'index (approximativement 2x la taille d'origine des données).

Avec ces informations, les lecteurs peuvent rapidement se rendre compte si le logiciel leur convient et envisager une implication dans le développement.

Avancement du développement

Les gens veulent toujours connaître la situation d'un projet. Pour les nouveaux projets, ils veulent apprécier l'écart existant entre les promesses et la réalité actuelle. Pour les plus avancés, ils cherchent à savoir comment est suivi le projet, le rythme de sorties des nouvelles versions et sa réactivité aux rapports de bogues, etc.

Pour répondre à ces questions vous devriez mettre en place une page d'avancement, listant les buts du projet à court terme et ses besoins (il pourrait, par exemple, être à la recherche de développeurs avec des compétences particulières). La page peut également fournir un historique des versions passées avec la liste de leurs fonctionnalités. Ainsi, les visiteurs peuvent se faire une idée de l'« avancement » du projet et sa vitesse de progression.

N'ayez pas peur de paraître non-préparé et ne cédez pas à la tentation d'exagérer la progression du développement. Chacun sait qu'un logiciel évolue par étapes, il n'y a pas de honte à dire « Ceci est une version alpha du logiciel avec des bogues connus. Il tourne et fonctionne au moins quelques fois, mais utilisez-le à vos propres risques ». Un tel langage ne fera pas fuir les développeurs nécessaires à cette étape. Quant aux utilisateurs, l'une des pires choses possible est de les attirer avant que le logiciel ne soit prêt. Une image d'instabilité ou de sensibilité aux bogues est très dure à gommer une fois bien ancrée. Le conservatisme est rentable sur le long terme, mieux vaut toujours un logiciel plus stable comparé aux attentes de l'utilisateur que le contraire : les bonnes surprises contribuent à un meilleur bouche à oreille.

Alpha et Bêta
Le terme alpha fait, en général, référence à une première version, avec laquelle il est possible de travailler, qui comporte toutes les fonctionnalités prévues, mais qui contient également des bogues connus.
Le but principal d'un logiciel alpha est de générer un retour, les développeurs sachant ainsi sur quoi travailler. La prochaine étape, bêta, veut dire que le logiciel a été épuré des bogues les plus sérieux mais n'a pas encore été suffisamment testé pour garantir une sortie.
Le but d'un logiciel bêta est : soit de devenir la version officielle, si aucun bogue n'est trouvé, soit d'apporter des critiques détaillées aux développeurs pour qu'ils puissent aboutir à la version finale rapidement. La différence entre alpha et bêta est principalement une question de jugement.

Téléchargements

Le logiciel doit être téléchargeable sous forme de code source aux formats standards. Au commencement d'un projet, les paquets binaires (exécutables) ne sont pas nécessaires, à moins que la complexité de l'assemblage demandé ou des dépendances soit telle que le simple fait de le faire fonctionner représente un gros travail pour la plupart des gens (mais si tel est le cas, le projet aura bien du mal à attirer les développeurs !).

Les mécanismes de distribution devraient être aussi pratiques, standards et transparents que possible. Pour éradiquer une maladie, vous ne distribueriez pas un médicament demandant une taille de seringue non standard pour être administré. De même, le logiciel devrait

se conformer aux règles des standards de construction et d'installation : plus il s'écartera des normes, plus les utilisateurs et développeurs potentiels abandonneront et partiront ailleurs, perplexes.

Cela peut sembler évident, mais de nombreux projets ne prennent pas la peine de standardiser les procédures d'installation avant un stade avancé du développement, se disant qu'ils peuvent le faire à n'importe quel moment. « On s'occupera de tout ça quand le code sera presque fini ». Ce qu'ils ne réalisent pas, c'est qu'en remettant le travail fastidieux de finition à plus tard, ils rendent en fait la finalisation du code plus longue car ils découragent les développeurs qui auraient pu autrement contribuer au code. C'est même plus insidieux que cela : ils ne savent pas qu'ils perdent tous leurs développeurs, ce processus étant l'accumulation de non-évènements. Par exemple : quelqu'un visite un site Web, télécharge le logiciel, essaye de l'assembler, échoue, abandonne et va voir ailleurs. Qui saura un jour que c'est arrivé, hormis l'individu en question ? Aucun acteur du projet ne réalisera que l'intérêt de quelqu'un et sa bonne volonté ont été silencieusement gâchés.

Le travail barbant à forte rentabilité devrait toujours être réalisé tôt. Faciliter significativement l'accès au projet en créant des paquets standards est un exemple de forte rentabilité.

Quand vous publiez un paquet téléchargeable, il est vital que vous lui attribuiez un numéro de version unique, ainsi les gens peuvent comparer deux versions et savoir laquelle est la plus récente. Une discussion détaillée à propos de la numérotation se trouve dans la section « Numérotation de version » et les détails à propos de la standardisation d'une construction et des procédures d'installation sont traités dans la section appelée « Création de Paquets », toutes les deux dans le chapitre 7.

Gestion de versions et accès au système de suivi de bogues

Télécharger les paquets sources est suffisant pour ceux qui veulent juste installer et utiliser un logiciel, mais c'est insatisfaisant pour ceux qui veulent le déboguer ou ajouter de nouvelles fonctionnalités. Des mises à jour chaque nuit peuvent aider, mais elles ne sont pas d'une précision assez fine pour une communauté de développeurs prospère. Les gens ont besoin d'un accès en temps réel aux dernières sources, et pour cela, on peut utiliser un programme de gestion de configuration logicielle. La présence de versions contrôlées du code source, anonymement accessibles, est un signe, à la fois pour les utilisateurs et les développeurs, que le projet fait un effort pour donner aux gens le nécessaire pour participer. Si vous ne pouvez pas proposer de gestion de versions tout de suite, alors affichez clairement votre intention de l'implanter rapidement. L'infrastructure d'une version de contrôle est discutée plus en détails dans la section nommée « Les logiciels de gestion de versions » dans le chapitre 3.

Il en va de même pour le système de suivi de bogues du projet. L'importance du suivi de bogues ne réside pas uniquement dans son utilité aux développeurs, mais aussi dans la signification qu'il a aux yeux des observateurs du projet. Pour la majorité, une base de données de bogues est l'un des signes les plus forts de la prise au sérieux du projet. Surtout : plus la base de données contient de bogues, meilleure est la santé du projet. Cela peut sembler illogique, mais souvenez-vous que le nombre de bogues enregistrés dépend principalement de trois choses : du nombre total de bogues présents, du nombre de personnes utilisant le

logiciel et de la facilité qu'ont ces utilisateurs à rapporter les nouveaux bogues. De ces trois facteurs, les deux derniers sont plus importants que le premier : comment le projet va-t-il gérer l'enregistrement et la gestion des priorités de ces bogues ? Un projet avec une base de données de bogues importante et développée (ce qui signifie que les bogues sont corrigés rapidement, que les bogues dupliqués sont unifiés, etc.) fait alors meilleure impression qu'un projet sans base de données de bogues ou avec une base presque vide.

Bien évidemment, si vous venez de débuter votre projet, votre base de données contiendra très peu de bogues ; il est difficile d'y faire grand-chose. Mais si la page d'avancement met l'accent sur la jeunesse du projet, et si ceux qui consultent la base de données de bogues peuvent voir que la majeure partie de l'archivage a été faite récemment, ils pourront en déduire que le projet a un rythme d'archivage sain et ne seront pas indûment alarmés par le nombre relativement faible de bogues enregistrés.

Remarquez que les systèmes de recherche de bogues ne sont souvent pas uniquement utilisés pour rapporter les bogues du logiciel mais aussi pour rapporter les demandes d'améliorations, de modifications de la documentation, des tâches en cours et plus encore. Les détails sur l'utilisation d'un système de suivi de bogues sont donnés dans la section « Suivi de bogues » dans le chapitre 3. S'étendre ici apparaît donc superflu mais l'aspect important, du point de vue de la présentation, est d'avoir un tel système de suivi, et de s'assurer que cela soit bien visible sur la page d'accueil du projet.

Les voies de communications

Les visiteurs veulent en général savoir comment contacter les êtres humains impliqués dans le projet. Fournissez les adresses des listes de diffusion, des canaux IRC et tout autre forum où chaque personne impliquée dans le logiciel peut être jointe. Indiquez clairement que vous-même et les autres auteurs du projets sont inscrits sur ces listes de diffusion afin que les gens puissent voir qu'il existe des moyens de donner leur avis aux développeurs. Votre présence sur les listes n'implique pas que vous êtes tenu de répondre à toutes les questions ou d'intégrer toutes les suggestions. Sur le long terme, la plupart des utilisateurs ne s'inscriront jamais aux forums de toute façon, mais ils seront rassurés de savoir qu'ils peuvent le faire si besoin est.

Au cours des premières étapes du projet, il n'est nul besoin de séparer les forums utilisateurs et développeurs, mais préférable de réunir toutes les personnes concernées par le logiciel dans une même « salle » pour en discuter. La distinction entre développeurs et utilisateurs est souvent floue pour les premiers arrivants, si tant est qu'elle existe : le nombre de développeurs par rapport à celui des utilisateurs est, en général, beaucoup plus élevé dans les premiers jours du projet que par la suite. Même si vous ne pouvez partir du principe que tout nouvel arrivant est un programmeur voulant bidouiller le logiciel, vous pouvez supposer qu'il est au minimum désireux de suivre les discussions autour du développement et de comprendre l'orientation du projet.

Comme ce chapitre ne traite que du lancement d'un projet, il est presque suffisant de dire que ces forums de communication doivent exister. Plus tard, dans la section appelée « Gérer la croissance » au chapitre 6, nous examinerons où et comment mettre en place de tels forums, dans quelle mesure ils peuvent avoir besoin d'une modération ou d'autres formes

de surveillance, et comment séparer, le moment venu, les forums utilisateurs des forums développeurs sans creuser de fossé infranchissable.

Les directives pour développeurs

Si une personne envisage de contribuer au projet, elle cherchera les directives pour développeurs. Celles-ci ne sont pas vraiment techniques mais plutôt sociales : elles expliquent comment les développeurs interagissent entre eux et avec les utilisateurs, et finalement comment les choses se déroulent.

Ce sujet est détaillé dans la section dénommée « Tout mettre par écrit » dans le chapitre 4, mais les éléments de base des directives pour les développeurs sont :

- des liens vers les forums pour l'interaction avec d'autres développeurs,
- des instructions pour les rapports de bogues et la soumission de correctifs,
- des indications sur la gestion du développement : le projet est-il une dictature bénévole, une démocratie ou quelque chose d'autre encore ?

Le mot « dictature » n'est pas employé dans un sens péjoratif. Une tyrannie où un développeur particulier a le droit de veto sur tous les changements n'a rien de choquant. Beaucoup de projets réussis fonctionnent de cette manière. Il est simplement important d'annoncer la couleur. Une tyrannie faisant semblant d'être une démocratie découragera les gens, une tyrannie qui ne se déguise pas fonctionnera très bien tant que le tyran est compétent et a la confiance de tous.

Voir svn.collab.net [1] pour un exemple particulièrement complet de directives pour les développeurs, ou openoffice.org [2] pour des directives plus larges qui se concentrent davantage sur l'administration et l'esprit de participation et moins sur les aspects techniques.

Le problème différent de la présentation du logiciel aux programmeurs est abordé dans la section nommée « Documentation développeur » plus loin dans ce chapitre.

Documentation

La documentation est essentielle. Il faut quelque chose à lire, même rudimentaire et incomplet. La documentation fait vraiment partie des « corvées » mentionnées précédemment, et c'est souvent le premier élément qui peut faire s'effondrer un nouveau projet Open Source. Proposer une mission et une liste de fonctionnalités, choisir une licence, résumer l'avancement du développement sont des tâches relativement simples pouvant être accomplies une bonne fois pour toutes et dont vous n'aurez plus à vous soucier. La documentation, à l'inverse, n'est jamais vraiment terminée, ce qui peut être parfois une bonne raison pour retarder son exécution.

Cependant, rédacteurs et lecteurs ne partagent pas les mêmes préoccupations. La documentation la plus importante pour les nouveaux utilisateurs concerne les bases : comment

1. http ://svn.collab.net/repos/svn/trunk/www/hacking.html
2. http ://www.openoffice.org/dev_docs/guidelines.html

rapidement mettre le logiciel en route, une vue d'ensemble de son fonctionnement et peut-être quelques guides pour les tâches courantes. Or, c'est exactement tout ce que les auteurs de la documentation connaissent parfaitement, si parfaitement, qu'il peut être difficile pour eux de voir les choses du point de vue du lecteur, et d'énumérer laborieusement les étapes qui (aux yeux des auteurs) semblent évidentes ou inutiles à mentionner.

Il n'y a pas de solution miracle à ce problème. Il faut prendre le temps d'écrire la documentation, puis de la mettre entre les mains d'un nouvel utilisateur lambda pour en tester la qualité. Utilisez un format facilement éditable comme le HTML, du texte brut, Texinfo ou une variante de XML, quelque chose de pratique en vue de retouches légères et d'améliorations à la volée. Il ne s'agit pas simplement de faciliter les choses aux contributeurs, mais aussi de permettre de travailler à ceux qui rejoindront plus tard le projet.

On peut s'assurer de réaliser la documentation initiale de base en limitant par avance son étendue. Ainsi, son écriture ne semblera pas être une tâche infinie. Par expérience, la documentation devrait remplir les critères minimaux suivant :

– Indiquer clairement au lecteur le degré d'expertise attendu de sa part.

– Décrire clairement et de manière détaillée comment lancer le logiciel et, quelque part en début de documentation, expliquer comment faire fonctionner une sorte de diagnostic ou une commande simple confirmant que tout a été fait correctement. Les instructions de démarrage sont d'une certaine manière plus importantes que les instructions d'utilisation. Plus une personne fournit d'efforts pour installer et faire ses premiers pas avec le logiciel, plus elle découvre les fonctionnalités avancées qui ne sont pas aussi bien documentées. Les abandons sont toujours rapides. Ce sont donc les premières étapes, comme l'installation, qui demandent le plus d'assistance.

– Présenter un exemple, ou une méthode de réalisation d'une tâche basique. Évidemment, de nombreux exemples seraient préférables, mais si vous êtes limité par le temps, sélectionnez une fonction et décrivez-la en détail. Dès qu'une personne constate que le logiciel peut servir à quelque chose, elle commence à explorer les autres possibilités voire, si vous êtes chanceux, compléter elle-même la documentation. Ce qui nous amène au point suivant…

– Indiquer les endroits où la documentation est incomplète. En montrant au lecteur que vous en connaissez les imperfections, vous vous accordez à son point de vue. Votre empathie le rassure : il n'a pas à lutter pour convaincre le projet de ce qui est important. Ces indications ne sont pas nécessairement des promesses de remédier à ces insuffisances à une date précise ; il est également légitime de les voir comme autant d'appels à des contributeurs volontaires.

Le dernier point est réellement de la plus haute importance et s'applique au projet entier, pas uniquement à la documentation. Tenir un compte précis des lacunes connues est la norme dans le monde de l'Open Source. Inutile d'exagérer les défauts du projet, il suffit simplement de les identifier scrupuleusement et impartialement quand le besoin s'en fait sentir (que ce soit dans la documentation, dans la base de données de suivi de bogues ou dans les discussions sur la liste de diffusion). Personne ne verra cela comme du défaitisme de la part du projet, ni comme une promesse de résolution des problèmes à une date précise, à moins

que le projet ne s'y engage explicitement. Étant donné que tout utilisateur du logiciel découvrira les lacunes de lui-même, il est préférable de le préparer psychologiquement, de plus, le projet donnera l'impression d'une parfaite connaissance de son état de fonctionnement.

Entretenir une FAQ
Une FAQ (« Frequently Asked Questions » ou « Foire Aux Questions ») peut être l'un des meilleurs investissements du projet pour son potentiel pédagogique. Une FAQ bien entretenue à de bonnes chances de répondre aux questions de ceux qui la consultent. La FAQ est souvent le premier document que les utilisateurs consultent lorsqu'ils ont un problème, souvent même avant le manuel officiel, et c'est sûrement le document de votre projet auquel les autres sites renverront.
Malheureusement vous ne pouvez pas créer la FAQ dès le début du projet. Les bonnes FAQ ne sont pas écrites, elles se façonnent. Ce sont, par définition, des documents réactifs, évoluant avec le temps en réponse à l'usage quotidien des utilisateurs du logiciel. Puisqu'il n'est pas possible d'anticiper toutes les questions que les gens peuvent poser, il est impossible de prendre cinq minutes de son temps et écrire une FAQ utile en partant de rien. Ne gaspillez donc pas votre temps à essayer de le faire. Vous pourriez, par contre, trouver utile de mettre en place un modèle de FAQ vide, pour que les gens trouvent facilement un endroit où écrire leurs questions et réponses une fois le projet sur les rails. Pour le moment, la chose la plus importante n'est pas son exhaustivité mais sa facilité d'utilisation : s'il est facile d'ajouter des questions/réponses à la FAQ, les gens le feront (l'entretien d'une bonne FAQ est une tâche peu aisée et fascinante, j'en reparlerai dans la section intitulée « Responsable FAQ » dans le chapitre 8).

Disponibilité de la documentation

La documentation devrait être disponible en deux endroits : en ligne, directement depuis le site Web, et dans la distribution téléchargeable du logiciel (voir la section appelée « Création de paquets » dans le chapitre 7). Elle doit être en ligne sous une forme navigable car les gens lisent souvent la documentation avant de télécharger le logiciel pour la première fois. Mais elle doit aussi accompagner le téléchargement du logiciel qui devrait fournir (c'est-à-dire rendre l'accès hors ligne possible) tout ce dont on pourrait avoir besoin pour utiliser le paquet.

En ce qui concerne la documentation en ligne, assurez-vous qu'elle comporte un lien vers la documentation complète sur une page au format HTML (ajoutez une note comme « monolithique », « tout en un » ou « page unique » à côté du lien, afin que les utilisateurs sachent qu'un certain temps est nécessaire pour télécharger). C'est utile car, souvent, ils cherchent un mot en particulier ou une phrase dans la documentation entière. En général, ils savent ce qu'ils veulent mais ne se souviennent pas où se trouve l'information. Pour eux, rien n'est plus frustrant que de rencontrer une page HTML pour le sommaire, puis une autre pour l'introduction, une autre encore pour les instructions d'installation, etc. Quand les pages sont séparées ainsi, la fonction de recherche du navigateur leur est inutile. Une structure éclatée est utile pour ceux qui savent à quelle partie se référer ou qui veulent lire le document du début à la fin par morceaux. Ne pas leur proposer un document complet sur une seule page leur compliquerait la vie plus qu'autre chose.

La documentation développeur

La documentation développeur est rédigée pour aider les programmeurs à comprendre le code et pouvoir le réparer et le compléter. Elle ne fait pas double emploi avec les directives pour les développeurs dont nous avons parlé précédemment, qui sont plus un outil social que technique. Ces consignes indiquent aux programmeurs comment bien collaborer entre eux, la documentation développeur leur indique comment se débrouiller avec le code lui-même. Les deux sont en général réunies en un seul document pour des raisons pratiques (comme dans l'exemple donné plus tôt : svn.collab.net [1]), mais ce n'est pas une obligation.

Bien que la documentation utilisateur puisse être très pratique, elle ne doit pas être la raison d'un retard de la distribution. Que les auteurs d'origine soient disponibles (et disposés à le faire) pour répondre aux questions touchant au code, est déjà suffisant pour commencer. En fait, c'est le fait de devoir répondre sans cesse aux mêmes questions qui motive souvent les gens à écrire la documentation. Mais, même avant qu'elle ne soit rédigée, les participants motivés trouveront comment fonctionne le code. En apprenant les bases du code, ils réalisent quelque chose d'utile pour eux, c'est ce qui les pousse à persévérer. Si les gens y croient vraiment, ils prendront le temps de comprendre les choses, s'ils n'ont pas cette foi, ce n'est pas la documentation, aussi complète puisse-t-elle être, qui les attirera ou les fera rester.

Donc, si vous n'avez que peu de temps à lui consacrer, écrivez la documentation pour les seuls utilisateurs. Au fond, la documentation utilisateur est également une documentation développeur puisque les programmeurs qui vont travailler sur une partie du logiciel doivent être familiers avec son utilisation. Ensuite, quand vous voyez que les programmeurs posent sans cesse les mêmes questions, prenez le temps d'écrire quelques documents indépendants plus spécialement pour eux.

Certains projets utilisent des wikis comme documentation de départ, voire comme documentation principale. D'après mon expérience, cela ne fonctionne vraiment que si le wiki est souvent édité par une poignée de gens en accord sur la manière d'organiser la documentation et sur le ton qu'elle devrait avoir. Référez-vous à la section « Wikis » dans le chapitre 3 pour en savoir plus.

Exemples de réalisations et captures d'écran

Si le projet comporte une interface graphique, s'il génère des réalisations visuelles quelconques, affichez-en quelques exemples sur le site du projet. Pour les interfaces, cela signifie des captures d'écran, pour les réalisations ça pourra prendre la forme de captures d'écrans ou simplement de fichiers. Les deux satisfont le besoin de gratification immédiate : une simple capture d'écran peut se montrer plus convaincante que des paragraphes de texte descriptif ou des conversations extraites de la liste de diffusion. Une capture d'écran est la preuve irréfutable que le logiciel fonctionne. Il peut être bogué, il peut être compliqué à installer, sa documentation n'est peut-être pas complète, mais cette capture d'écran est la preuve que, si on y met assez de bonne volonté, on peut le faire fonctionner.

1. http ://svn.collab.net/repos/svn/trunk/www/hacking.html

Captures d'écran
Les captures d'écran peuvent être intimidantes tant que vous n'en avez pas réellement pris quelques-unes. Voici quelques instructions pour en réaliser. En utilisant « the Gimp » (gimp.org), ouvrez Fichiers > Acquisition > Capture d'écran, choisissez Fenêtre simple ou Écran entier puis faites Ok. Maintenant votre prochain clic de souris prendra une photo de la fenêtre ou de l'écran sur lequel vous cliquez sous la forme d'une image dans « the Gimp ». Coupez et re-dimensionnez l'image si nécessaire, et au besoin, servez-vous des tutoriels disponibles sur le site.

Vous pouvez ajouter bien d'autres choses à votre site Web, si vous en avez le temps ou si, pour une raison ou pour une autre, elles vous apparaissent particulièrement appropriées : une page d'informations, une page pour l'historique du projet, une page pour les liens connexes, une fonction de recherche, un lien pour les dons, etc. Rien de tout ceci n'est absolument nécessaire au moment du lancement, mais gardez-les à l'esprit pour la suite.

Forges

Quelques sites proposent un hébergement gratuit ainsi que l'infrastructure pour des projets Open Source : une zone Web, une gestion de versions, un système de suivi de bogues, une zone de téléchargement, des forums de discussions, des sauvegardes régulières, etc. Les détails varient d'un site à l'autre, mais les services de base sont les mêmes partout. En utilisant l'un de ces sites vous obtiendrez beaucoup en échange de rien. Vous tournez par contre le dos à un contrôle précis de ce que vous offrez aux utilisateurs du site. Les services d'hébergement décident des logiciels qui fonctionnent sur le site et peuvent contrôler, ou au moins influencer, l'aspect des pages Web du projet.

Je vous renvoie à la section « Forges » dans le chapitre 3 pour un exposé plus détaillé des avantages et inconvénients des forges ainsi qu'une liste de sites qui offrent ces services.

2. Choisir une licence et l'appliquer

Cette partie est un guide rapide, non exhaustif, du choix d'une licence. Lisez le chapitre 3 pour mieux comprendre les détails des implications légales des différentes licences et l'influence qu'aura votre choix sur l'intégration de votre logiciel dans d'autres projets libres.

Vous avez le choix entre un grand nombre de licences. Nous n'avons pas besoin de parler de la plupart d'entre elles ici puisqu'elles ont été écrites pour satisfaire des besoin légaux très particuliers pour certaines sociétés ou personnes et qu'elles ne seraient pas appropriées pour votre projet. Nous nous en tiendrons simplement aux licences les plus courantes, dans la plupart des cas votre choix se portera sur l'une d'elles.

Les licences à « tout faire »

Si cela ne vous dérange pas que le code de votre projet puisse être utilisé dans un programme propriétaire vous devriez utiliser une licence du type MIT/X. C'est la plus simple parmi certaines licences minimales qui ne font pas grand chose de plus que revendiquer un droit d'auteur nominal (sans pour autant limiter la copie) et spécifier que le code est livré sans garantie. Référez-vous à la section « Le système de licence MIT/X Window » pour plus de détails.

La licence GPL

Si vous ne souhaitez pas que votre code puisse être utilisé dans des programmes propriétaires utilisez la licence GNU General Public Licence [1]. La licence GPL est certainement la licence pour logiciel libre la plus largement reconnue dans le monde aujourd'hui, ce qui est déjà un avantage important puisque les utilisateurs et les bénévoles seront familiers avec cette licence et donc n'auront pas à passer du temps en plus pour la lire et la comprendre. Voir la section nommée « La licence GNU General Public License » dans le chapitre 9 pour de plus amples informations.

Si les utilisateurs font principalement usage de votre code sur le réseau, c'est à dire que votre logiciel est principalement exploité au sein d'un service en ligne, vous devriez plutôt appliquer la licence GNU Affero GPL. Référez-vous à la partie « La GNU Affero GPL : une version de la GNU GPL pour le code côté serveur » au chapitre 9.

Comment appliquer une licence à votre logiciel

Une fois la licence choisie vous devriez la faire apparaître sur la première page du projet. Vous n'avez pas besoin d'ajouter le texte de la licence à cet endroit : précisez simplement le nom de la licence et créez un lien vers le texte complet se trouvant sur une autre page.

Vous annoncez ainsi au public sous quelle licence vous comptez distribuer le projet, mais ce n'est pas légalement suffisant. Pour ceci il faut que le logiciel contienne cette licence. Habituellement, le texte complet de la licence se trouve dans un fichier nommé *copie* (ou *licence*), et une petite note en début de chaque fichier source indique la date du copyright, le détenteur des droits et l'emplacement du texte complet de la licence.

Il y a beaucoup de variantes possibles à ce modèle, nous verrons donc juste un exemple ici. La licence GNU GPL recommande d'ajouter une note comme celle ci-dessous au début de chaque fichier source.

```
Copyright (C) year  name of author
This program is free software: you can
```

1. http ://www.gnu.org/licences/gpl.html

```
redistribute it and/or modify
it under the terms of the GNU General
Public License as published by
the Free Software Foundation,
either version 3 of the License, or
(at your option) any later version.
This program is distributed in the hope
that it will be useful,
but WITHOUT ANY WARRANTY;
without even the implied warranty of
MERCHANTABILITY or FITNESS FOR
A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.
You should have received a copy of the GNU
General Public License
along with this program.  If not,
see http://www.gnu.org/licenses/
```

Ou en français :

```
Copyright (C) année nom de l'auteur
Ce programme est un logiciel libre : vous pouvez
le redistribuer et/ou le modifier sous les termes
de la licence GNU Public Licence telle que publiée
par la Free Software Foundation, soit dans la
version 2 de la licence, ou (selon votre choix)
toute version ultérieure.
Ce programme est distribué avec l'espoir
qu'il sera utile, mais SANS AUCUNE
GARANTIE : sans même les garanties
implicites de VALEUR MARCHANDE ou
D'APPLICABILITÉ À UN BUT PRÉCIS. Voir
la licence GNU General Public License
pour plus de détails.
Vous devriez avoir reçu une copie de la licence
GNU General Public Licence
avec ce programme. Si ce n'est pas le cas,
voir http://www.gnu.org/licenses/
```

Rien n'indique spécifiquement que la licence livrée avec le programme est dans le fichier « COPIE », mais c'est là qu'elle est en général copiée (vous pouvez modifier la note ci-dessus pour le préciser directement). Ce modèle fournit également l'adresse postale où demander une copie de la licence. Une autre méthode courante est de fournir un lien vers

le site Web où se trouve la licence. Fiez-vous à votre jugement et renvoyez à la page que vous pensez être la plus à même d'être maintenue, ce qui peut simplement être le site de votre projet. En général la note que vous ajoutez dans chaque fichier source n'a pas besoin de ressembler exactement à ceci, tant qu'elle commence en indiquant le détenteur des droits et la date, précise le nom de la licence et dit clairement où la licence peut être trouvée.

3. Donner le ton

Jusque-là, nous avons vu les tâches à faire une fois pour toutes à la mise en route : choisir une licence, créer le site Web, etc. Mais les aspects les plus importants de la création d'un projet changent sans cesse. Le choix d'une adresse pour la liste de diffusion est simple, vous assurer que les discussions sur cette liste ne dérivent pas est complètement différent. Si le projet s'ouvre après des années de développement interne, son processus de développement s'en retrouvera modifié et vous devrez préparer les développeurs déjà présents à ce changement.

Les premiers pas sont les plus durs car les modèles et les attentes pour le futur n'ont pas encore été définis. La stabilité d'un projet n'est pas due à des règles formelles mais à une sagesse collective, difficile à définir, qui se développe avec le temps. On retrouve souvent des règles écrites également, mais elles représentent plutôt une version épurée des accords impalpables et toujours fluctuants qui dirigent vraiment le projet. Les règles écrites ne proposent pas une définition du projet mais plutôt sa description, qui est assez approximative d'ailleurs.

Il y a plusieurs raisons à ce mode de fonctionnement. La croissance du projet et un fort taux de renouvellement de ses participants ne perturbent pas tellement les règles sociales. Tant que les changements ne se produisent pas trop vite, les nouveaux arrivants ont le temps d'assimiler les règles du jeu, et après avoir les avoir apprises, ils contribuent à les perpétuer. Voyez comme les chansons enfantines traversent les siècles. Les enfants d'aujourd'hui chantent plus ou moins les mêmes qu'il y a des centaines d'années. Les jeunes enfants entendent des chansons chantées par leurs aînés. Puis, en devenant adultes, ils les chantent à leur tour devant de plus jeunes. Les enfants n'entrent évidemment pas consciemment dans ce processus de transmission, mais les chansons survivent parce qu'elles sont malgré tout transmises de manière régulière et répétitive. L'échelle de temps des logiciels libres ne se mesure peut-être pas en siècles (on ne sait pas pour l'instant), mais les moyens de diffusion sont très proches. Le taux de renouvellement est plus rapide cependant, et doit être compensé par un effort de transmission plus actif et volontaire.

Cet effort est aidé par le fait que les gens arrivent en cherchant et en espérant trouver des règles du jeu. L'être humain est ainsi fait. Dans un groupe unifié par un but commun, les recrues chercheront instinctivement à montrer par leurs attitudes qu'elles font partie du groupe. Le but est d'établir rapidement des modèles, et de rendre ces comportements « de groupe » utiles au projet, car une fois en place ils se perpétueront principalement d'eux-mêmes.

Vous trouverez ci-dessous quelques pistes qui vous guideront vers la bonne marche à suivre. Cette liste n'est pas exhaustive, ce sont simplement des illustrations de l'idée que la

création précoce d'une bonne collaboration aide énormément le projet. Physiquement, les développeurs travailleront peut-être seuls, mais vous pouvez faire en sorte qu'ils se sentent plus proches, comme s'ils travaillaient tous dans la même pièce. Si vous réussissez, ils auront envie de passer plus de temps sur le projet. Je choisis ces exemples particuliers car le projet Subversion y a été confronté [1]. J'y ai participé et ai observé, depuis le tout début, ces problèmes. Mais ils ne sont pas spécifiques à Subversion, la plupart des projets Open Source les rencontreront et ces exemples doivent être pris comme des opportunités de commencer du bon pied.

Évitez les discussions privées

Une fois le projet ouvert au public, vous et les autres fondateurs serez tentés de régler les problèmes délicats par des discussions privées au sein d'un cercle fermé. C'est particulièrement vrai dans les premiers jours, quand il y a tant de décisions importantes à prendre et, généralement, peu de volontaires qualifiés pour le faire. Tous les inconvénients des discussions publiques vous apparaîtront de manière palpable : le délai inhérent aux discussions par e-mail, le besoin de laisser suffisamment de temps à un consensus pour se former, le tracas d'avoir affaire aux volontaires naïfs qui pensent saisir les tenants et les aboutissants du problème alors que ce n'est pas le cas (tout projet a cette catégorie de volontaires, ils pourront très bien devenir les membres les plus actifs l'année suivante, tout comme ils peuvent aussi rester naïfs pour toujours), l'ennui d'avoir affaire à la personne qui ne comprend pas pourquoi vous voulez résoudre juste le problème X alors qu'il fait partie de manière évidente d'un plus grand problème Y et ainsi de suite. La tentation sera effectivement grande de prendre les décisions en petit comité et de les présenter comme des faits accomplis, ou au moins comme les recommandations d'une grande partie soudée et influente des votants.

Ne le faites pas.

Aussi encombrantes et lentes qu'elles puissent être, les discussions publiques seront toujours préférables sur le long terme. Prendre les décisions importantes en privé reviendrait à pulvériser du *« repousse-bénévoles »* sur le projet. Aucun volontaire sérieux ne resterait longtemps dans un environnement où toutes les grandes décisions sont prises par un conseil privé. De plus, les discussions publiques ont des effets secondaires positifs qui perdureront bien après l'éphémère question technique débattue :

- La discussion aidera à entraîner et instruire les nouveaux développeurs. Vous ne savez jamais combien de paires d'yeux suivent une conversation, même si la plupart ne participent pas, ils peuvent être nombreux à la suivre en silence, glanant des informations sur le logiciel.
- La discussion vous entraînera dans l'art d'expliquer les problèmes techniques aux gens qui ne sont pas aussi familiers que vous avec le logiciel. C'est une compétence qui demande de la pratique, ce que vous ne pouvez pas acquérir en parlant avec des gens qui savent déjà ce que vous savez.
- La discussion et ses conclusions seront ensuite accessibles en permanence dans des archives publiques, permettant aux discussions futures d'éviter de tourner en rond. Voir la section « Utilisation efficace des archives » dans le chapitre 6.

1. http ://subversion.tigris.org/

Finalement, il se peut que quelqu'un dans la liste apporte une vraie contribution à la discussion, en proposant une idée à laquelle vous n'auriez pas pensé. Il est difficile d'en déterminer la probabilité : ça dépend simplement de la complexité du code et du degré de spécialisation requis. Mais s'il m'est accordé de fournir une preuve anecdotique, je me risquerais à dire que la possibilité en est très forte. Dans le projet Subversion, nous (les fondateurs) pensions que nous faisions face à un ensemble de problèmes graves et complexes auxquels nous réfléchissions beaucoup depuis quelques mois. Nous doutions franchement qu'un membre de la nouvelle liste de diffusion puisse apporter une vraie contribution à la discussion. Alors, nous avons employé la manière simple et nous avons commencé à débattre d'idées techniques par e-mails privés, jusqu'à ce que quelqu'un qui observait le projet [1] ait eu vent de ce qui se passait et ait demandé à ce que la discussion soit partagée dans la liste publique. Nous l'avons fait, un peu mécontents, et nous avons été surpris par le nombre de commentaires inspirés et de suggestions qui en ont rapidement résulté. Dans beaucoup de cas, les gens proposaient des idées qui ne nous étaient pas venues à l'esprit. En fin de compte, il y avait des gens très malins sur cette liste, ils attendaient simplement qu'on leur tende la main. J'avoue que les discussions qui ont suivi ont pris plus de temps qu'elles ne l'auraient fait si nous avions gardé cette conversation privée, mais elles étaient tellement plus productives que le temps n'était pas perdu, loin s'en faut.

Sans tomber dans les généralisations faciles comme « le groupe est toujours plus fort que l'individu » (nous avons tous connus suffisamment de groupes pour savoir que ce n'est pas forcément vrai), il faut reconnaître que le groupe excelle dans certains domaines. Une inspection à grande échelle par nos confrères en fait partie, générer un grand nombre d'idées rapidement également. La qualité des idées dépend de la qualité de la réflexion menée bien sûr, mais vous ne pourrez jamais jauger le niveau de vos collaborateurs si vous ne leur soumettez pas un problème difficile.

Évidemment, certaines discussions doivent rester privées, nous en verrons des exemples tout au long du livre. Mais le principe directeur devrait toujours être : *s'il n'y a pas de raison qu'elle soit privée alors elle devrait être publique*.

Rendre cela possible demandera des efforts. Il ne suffit pas simplement de s'assurer que tous vos billets passent par la liste publique. Vous devez aussi faire en sorte que les discussions entre les autres membres qui n'ont pas de raison d'être privées figurent également sur la liste publique. Si quelqu'un tente de commencer une conversation privée, sans raison de confidentialité particulière, il vous incombe alors de rendre public l'embryon de discussion au plus tôt. Ne faites pas de commentaires sur le sujet d'origine avant d'avoir réussi à le diriger vers un endroit public ou de vous être assuré que la confidentialité était requise. Si vous insistez, les gens comprendront rapidement et commenceront à utiliser les forums publics naturellement.

1. Nous n'en sommes pas encore aux remerciements, mais je vais déjà mettre en pratique ce que je prêcherai plus tard : l'observateur était Brian Behlendorf et c'est lui qui a signalé l'importance de maintenir toutes les discussions publiques à moins que vous n'ayez un besoin particulier de confidentialité.

Tuez la vulgarité dans l'oeuf

Dès le début de votre projet, vous devriez maintenir une politique de tolérance zéro contre les comportements grossiers et insultants au sein des forums. La tolérance zéro n'implique pas l'emploi des grands moyens. Vous n'êtes pas censé bannir des personnes des listes de diffusion parce qu'elles s'en sont pris à un autre membre, ni leur retirer leur accès de *commit* pour des commentaires désobligeants (en théorie vous serez peut-être amené à vous y résoudre, mais seulement après avoir épuisé toutes les autres solutions, ce qui par nature n'est pas plausible au début du projet). Montrer une tolérance nulle signifie simplement ne jamais fermer les yeux sur un mauvais comportement. Par exemple, quand quelqu'un publie un commentaire technique mêlé à une attaque *ad hominem* contre un autre développeur travaillant sur le projet, vous devez en premier lieu répondre à l'attaque comme à un problème différent et, seulement ensuite, revenir au contenu technique du message.

Il est malheureusement très facile, et bien trop courant, que des discussions constructives sombrent en batailles rangées. Les gens peuvent dire par e-mail des choses qu'ils n'oseraient jamais se dire en face. Le sujet de discussion fait caisse de résonance : concernant les problèmes techniques, les gens pensent souvent qu'une seule bonne solution répond à la plupart des questions et qu'un avis divergent ne peut s'expliquer que par l'ignorance ou la stupidité. Il n'y a qu'un pas entre dire que la proposition d'une personne est stupide et dire que la personne elle-même est stupide. En fait, la frontière entre débat technique et attaque personnelle est souvent difficile à discerner. C'est l'une des raisons pour lesquelles les mesures drastiques ou les punitions ne sont pas une bonne idée. Quand vous pensez être témoin de ceci, faites plutôt une réponse incitant les gens à conserver un ton amical dans les discussions, sans accuser personne d'avoir été délibérément fauteur de troubles. Ces réponses très « policier sympa » ont une malheureuse tendance à ressembler à un cours sur les bonnes manières donné par un maître de maternelle :

> *« Pour commencer, occupons nous, si vous le voulez bien, des commentaires potentiellement ad hominem ; par exemple, dire que le modèle de J pour la couche de sécurité est « naïf et ignorant des principes de base de la sécurité informatique » n'est en aucun cas une manière d'en débattre, que cela soit vrai ou pas. J a fait sa proposition en toute bonne foi. Si elle présente des inconvénients, indiquez-les et on les corrigera ou on décidera de pencher pour une autre solution. Je suis sûr que ce n'était pas l'intention de M d'insulter personnellement J, mais les mots étaient mal choisis et on essaie de garder les discussions constructives ici.*
>
> *En ce qui concerne la proposition. Je pense que M a raison lorsqu'il dit que… »*

Bien que des réponses de ce genre semblent artificiellement formelles, elles ont un effet visible. Si vous montrez sans cesse du doigt les mauvais comportements, sans demander d'excuses ou de reconnaissance de faute à la personne offensante, les gens se calmeront doucement et montreront un meilleur visage la prochaine fois en se comportant de manière plus courtoise. Un des secrets pour que cette technique soit payante est de ne jamais laisser le hors-sujet devenir la discussion principale. Ça doit rester une parenthèse, comme une

courte préface au corps de votre réponse. Indiquez en passant que les choses ne se déroulent pas ainsi par ici, mais ensuite, passez au vrai sujet afin de recentrer le débat et de donner aux autres matière à répondre. Si quelqu'un pense ne pas avoir mérité votre reproche et conteste, refusez simplement d'être attiré dans une dispute à ce propos. Vous pouvez ne pas répondre (si vous estimez qu'il se défoule simplement et que ça n'appelle pas de réponse) ou dire que vous êtes désolé si vous avez réagi excessivement et qu'il est difficile de détecter les nuances par e-mail, ensuite revenez-en au sujet principal. N'insistez surtout jamais pour qu'une personne qui s'est mal conduite reconnaisse ses torts, que ce soit publiquement ou en privé. S'ils décident de leur propre chef de faire un message d'excuse, c'est très bien, mais leur demander de le faire apportera uniquement de la rancœur.

Il importe surtout que les gens voient l'usage des bonnes manières comme un comportement du « groupe ». C'est important pour le projet puisque les développeurs peuvent être découragés (même au sein d'un projet qu'ils aiment et veulent soutenir) par des guerres d'insultes. Ça pourrait même arriver sans que vous ne vous en rendiez compte, quelqu'un pourrait jeter un œil aux discussions, voir qu'il faut la peau dure pour participer à ce projet et décide finalement de ne pas s'impliquer. Maintenir les forums accueillants est une stratégie de survie à long terme, et c'est plus simple à réaliser quand le projet est encore petit. Une fois que ça sera entré dans les mœurs, vous ne serez plus le seul à insister sur la politesse, tout le monde le fera.

Effectuez une inspection visible du code

L'un des meilleurs moyens de motiver une communauté de développement productive est d'amener les gens à regarder le code des autres. Une certaine infrastructure technique est requise pour le faire efficacement, en particulier les e-mails de *commit* doivent être activés, voir la section « E-mails de *commit* » pour plus de détails. Chaque fois que quelqu'un modifie le code source, un e-mail est ainsi envoyé montrant le journal et les *diffs* concernant les changements (voir *diff* dans la section nommée « Vocabulaire de la gestion de versions »). L'inspection du code consiste à étudier les e-mails de *commit* à mesure qu'ils arrivent, à chercher des bogues et les améliorations possibles [1].

L'inspection du code a plusieurs utilités. C'est l'exemple le plus probant de critique par les pairs dans le monde de l'Open Source, il aide directement à maintenir la qualité du logiciel. Un bogue a pu être soumis en même temps qu'une partie du logiciel sans être détecté, donc plus il y a d'yeux qui surveillent les *commits*, moins il y a de bogues passant au travers des mailles du filet. Mais l'inspection du code est aussi utile de manière indirecte : elle confirme que ce que font les membres compte pour quelque chose, en effet personne ne prendrait le temps d'inspecter un *commit* si c'était en pure perte. Les gens donnent le meilleur d'eux-mêmes quand ils savent que d'autres prendront le temps d'évaluer leur travail.

Les inspections devraient être publiques. Même lorsque j'étais dans la même salle que les développeurs et que l'un de nous avait apporté une contribution, nous n'en faisions pas la critique verbalement sur place mais l'envoyions plutôt à la liste de diffusion des développeurs.

1. C'est en général comme ça que se passe l'inspection du code dans les projets Open Source dans tous les cas. Dans des projets plus centralisés, « inspection du projet » peut aussi désigner une assemblée de gens s'asseyant autour d'une table pour relire des copies du code source, à la recherche de problèmes et de formules particulières.

C'est dans l'intérêt de chacun de voir que la critique a été faite. Les gens s'intéressent aux commentaires, parfois ils y trouvent des défauts, mais, même s'ils n'y participent pas, cela leur rappelle que la critique régulière est attendue : cela doit devenir une activité constante comme faire la vaisselle ou tondre le gazon.

Dans le projet Subversion, nous n'avons pas fait au départ l'exercice d'inspection régulière du code. Nous n'avions aucune garantie que tous les *commits* seraient revus, même si quelqu'un pouvait de temps en temps examiner un changement s'il était particulièrement intéressé par cette partie du code. Des bogues nous ont échappé qui auraient vraiment pu et auraient dû être vus. Un développeur, du nom de Greg Stein, qui connaissait l'importance de l'inspection du code par ses expériences passées, a décidé de montrer l'exemple en inspectant chaque ligne de chaque *commit* destiné au dépôt. Tout *commit* effectué était rapidement suivi d'un e-mail envoyé à la liste des développeurs par Greg, disséquant le *commit*, analysant les problèmes possibles et parfois faisant l'éloge d'un morceau de code bien écrit. Il détectait tout de suite les bogues et les habitudes non optimales d'écriture de code qui nous auraient autrement échappé. Évidemment, il ne s'est jamais plaint d'être le seul à inspecter absolument tous les *commits*, même si ça lui prenait une grande partie de son temps, mais il faisait l'éloge de l'inspection de code à chaque fois qu'il en avait l'occasion.

Rapidement, d'autres personnes, y compris moi-même, ont aussi commencé à inspecter les *commits* de façon régulière. Quelle était notre motivation ? Ce n'est pas Greg qui nous avait donné mauvaise conscience. Non, il nous a prouvé que passer du temps à inspecter le code est payant, et que quelqu'un peut apporter autant au projet en examinant les modifications des autres, qu'en écrivant de nouvelles lignes de code. Une fois ceci démontré, ce processus est devenu la norme, à tel point que lorsqu'un *commit* ne recevait pas de commentaires, son auteur s'en inquiétait, et demandait même sur la liste si quelqu'un avait eu l'occasion de le contrôler. Plus tard, Greg trouva un emploi ne lui laissant plus autant de temps pour Subversion et dut arrêter ses inspections régulières. Mais à ce moment là, l'habitude était tellement ancrée parmi nous, qu'elle semblait présente depuis des temps immémoriaux.

Commencez les examens dès le tout premier *commit*. Les problèmes les plus simples à détecter en fouillant les *diffs* sont les vulnérabilités de sécurité, les fuites de mémoire, les commentaires ou la documentation API insuffisants, les erreurs de logique, les incohérences de discipline appelant/appelé et d'autres problèmes qui nécessitent peu de contexte pour être détectés. Néanmoins, même des problèmes à plus grande échelle, comme l'incapacité à réduire des schémas répétés à une seule position deviennent détectables une fois que la personne fait des inspections régulièrement, parce que se souvenir des *diffs* antérieurs aide à l'inspection des *diffs* plus récents.

Ne vous inquiétez pas si vous ne trouvez rien à commenter ou si vous n'êtes pas assez renseigné sur toutes les parties du code. Il y aura presque toujours quelque chose à dire à propos de chaque *commit*, même si vous ne trouvez rien à redire, vous aurez peut-être un commentaire élogieux à faire. Le plus important est de montrer à ceux qui envoient les *commits* que ce qu'ils font est vu et compris. Évidemment, l'inspection du code ne dispense pas les programmeurs de la responsabilité de contrôler et tester leurs changements avant de les exposer : on ne devrait pas se reposer sur l'inspection du code pour trouver les erreurs qu'on aurait dû voir soi-même.

Soyez attentifs à l'amplitude des changements lorsque vous libérez un projet fermé

Si vous libérez un projet déjà existant, sur lequel les développeurs sont habitués à travailler dans un environnement fermé, assurez-vous que tout le monde comprenne qu'un grand changement se profile, essayez de vous mettre à leur place pour comprendre leurs sentiments.

Essayez d'imaginer la situation de leur point de vue : avant, toutes les décisions concernant le code et l'architecture étaient prises au sein d'un groupe où tous les programmeurs connaissaient à peu près aussi bien le logiciel, programmeurs qui étaient sous la même pression du même encadrement et qui connaissaient les forces et les faiblesses de chacun. Maintenant vous leur demandez d'exposer le code au regard d'étrangers qui se feront un jugement uniquement d'après ce qu'ils voient, sans connaître les pressions commerciales derrière telle ou telle décision. Ces étrangers poseront beaucoup de questions, des questions qui seront comme un choc pour les développeurs déjà présents qui réaliseront que la documentation sur laquelle ils se sont échinés est encore inadaptée (c'est inévitable). Pire encore, les nouveaux venus sont des inconnus, des entités sans visage. Si l'un de vos développeurs a déjà des doutes quant à ses compétences, imaginez comment ce sentiment sera exacerbé quand des nouveaux venus montreront du doigt des défauts du code qu'il a écrit, et pire encore, devant tous ses collègues. À moins que votre équipe soit composée de codeurs parfaits, c'est inévitable, en fait, ça leur arrivera à tous au début. Ce n'est pas parce qu'ils sont de mauvais programmeurs, c'est juste que n'importe quel programme ayant atteint une certaine taille contient des bogues et la critique par des pairs mettra en avant certains de ces bogues (voir la section appelée « Effectuez une inspection visible du code », précédemment dans ce chapitre). À ce moment, les nouveaux venus ne seront pas exposés à la critique des pairs, comme ils ne peuvent pas participer à la programmation tant qu'ils ne sont pas familiers avec le projet. Vos développeurs auront l'impression que la critique ne va que dans un sens. Il y a donc un risque qu'une mentalité d'assiégé se développe au sein des « anciens ».

Le meilleur moyen d'éviter ceci, est de tous les prévenir de ce qui les attend, expliquez, dites-leur que l'inconfort du début est parfaitement normal et assurez-les que les choses iront en s'améliorant. Certains de ces avertissements devraient être faits en privé, avant que le projet ne soit ouvert. Mais vous pourrez aussi trouver utile de rappeler aux gens sur les listes publiques que c'est une nouvelle phase du développement du projet et qu'il y aura un certain temps d'adaptation. La meilleure chose que vous puissiez faire est de donner l'exemple. Demander aux développeurs de répondre plus aux questions de débutants, si vous vous rendez compte qu'ils ne le font pas suffisamment, n'améliorera pas les choses. Ils n'ont pas forcément déjà une idée claire des questions qui requièrent une réponse ou pas, ou il se peut qu'ils ne sachent pas comment répartir leur temps entre le travail de programmation et leur nouvelle charge de communication externe. Afin de les faire participer, il faut que vous participiez vous-même. Soyez présent sur les listes de diffusion publiques et répondez à quelques questions posées par ce biais. Si vous n'avez pas la connaissance requise pour répondre à une question, alors transmettez-la à un développeur qui saura y répondre, et ce de manière visible à tous. Assurez-vous qu'il fournit une solution, ou qu'au moins il répond. Il sera tentant pour les développeurs qui ont participé de longue date de retomber dans des discussions privées, c'est naturel puisqu'ils y sont habitués. Assurez-vous de faire partie de

la liste de diffusion interne sur laquelle cela peut se passer, ainsi vous pouvez demander à ce que ce genre de discussion soit déplacée sur les listes publiques immédiatement.

Il y a d'autre préoccupations sur le long terme lorsque vous rendez public un projet précédemment fermé. Le chapitre 5 explore des techniques pour faire cohabiter avec succès des développeurs salariés et bénévoles et le chapitre 9 traite de l'attention légale que vous devez porter lorsque vous rendez libre un code propriétaire pouvant contenir des logiciels écrits ou « détenus » par d'autres parties.

4. Annoncer

Une fois que le projet est présentable, pas parfait, juste présentable, vous êtes prêt à l'annoncer au monde entier. C'est en fait un processus très simple : allez sur freshmeat.net [1], cliquez sur « Submit » (Soumettre) en haut de la barre de navigation et remplissez un formulaire pour annoncer l'existence de votre nouveau projet. C'est sur Freshmeat que tout le monde se rend pour surveiller les annonces concernant les nouveaux projets. Vous devrez attirer l'attention de quelques paires d'yeux ici pour que le bouche à oreille à propos de votre projet commence.

Si vous connaissez des listes de discussion ou des newsgroups où l'annonce de votre projet cadrerait avec le sujet et apporterait quelque chose alors écrivez-y. Mais prenez garde de faire exactement un sujet par forum et de rediriger les gens vers le forum de votre projet pour poursuivre la discussion (en modifiant *Répondre à* dans l'en-tête). Le message se doit d'être court et d'aller directement au but :

```
À: discuss@lists.example.org
Sujet: [ANN] Projet Scanley d'indexation
de textes complets
Répondre à: dev@scanley.org

Ceci est un message ponctuel pour annoncer
la création du projet Scanley, un outil
d'indexation de texte complet, Open
Source, ainsi qu'un moteur de recherche
avec une API riche, à l'usage des
programmeurs afin de permettre un service de
recherche pour de grandes quantités de
documents écrits.
Scanley est fonctionnel, en cours de
développement intense et recherche des
développeurs et des testeurs.
Site internet: http://www.scanley.org

Fonctionnalités:
```

1. http ://freshmeat.net/

```
* Recherche de texte brut, HTML et XML
* Recherche de mot ou de phrase
* (prévu) Correspondance approximative
* (prévu) Mise à jour incrémentale des index
* (prévu) Indexation de sites Web distants

Pré-requis:
* Python 2.2 ou supérieur
* Suffisamment d'espace disque pour
contenir les index (approximativement
2x la taille d'origine des données)

Pour de plus amples informations,
visitez scanley.org.

Merci,
- A. Nonyme
```

(Voir la section appelée « Publicité » dans le chapitre 6, vous y trouverez des conseils concernant l'annonce des futures versions et autres évènements liés au projet.)

Il y a un débat actuellement dans le monde du logiciel libre : doit-on commencer avec un code fonctionnel ou est-il plus profitable pour le projet d'être rendu public même pendant la période de conception / discussion ? Auparavant je pensais que commencer un projet avec un code qui marche était le facteur le plus important, que c'était ce qui séparait les projets réussis des gadgets et que les développeurs sérieux ne seraient attirés que par des logiciels qui réalisent déjà quelque chose de concret.

Finalement ce n'est pas toujours le cas. Pour le projet Subversion nous avons commencé avec un modèle, un cœur de développeurs intéressés et proches, beaucoup de paillettes et absolument aucun code. À ma grande surprise, le projet a rassemblé des participants actifs dès le début, et au moment où nous avions quelque chose de fonctionnel, il y avait déjà quelques volontaires très impliqués dans le projet. Subversion n'est pas le seul exemple, le projet Mozilla a également commencé sans code fonctionnel, et maintenant c'est un navigateur Internet populaire qui rencontre beaucoup de succès.

Face à de telles preuves, j'ai dû revoir ma position sur le fait qu'un code fonctionnel est absolument nécessaire pour lancer un projet. Un code fonctionnel reste la meilleure base du succès, et l'expérience voudrait qu'on attende de l'avoir avant d'annoncer la création du projet. Quoiqu'il en soit, certaines circonstances font qu'une annonce anticipée a du sens. Je pense en effet qu'un modèle bien fait, ou une sorte de trame pour le code est nécessaire. Évidemment il peut être retouché en fonction des critiques publiques, mais il faut proposer quelque chose de concret à quoi les gens pourront se raccrocher, quelque chose de plus tangible que de bonnes intentions.

Chaque fois que vous annoncez, ne vous attendez pas à voir débarquer ensuite une horde de volontaires désirant rejoindre le projet. En général, le résultat de votre annonce sera la réception de quelques questions désinvoltes, quelques personnes supplémentaires joindront les

listes de diffusion et, à part cela, les choses continueront comme avant. Mais avec le temps, vous remarquerez une augmentation progressive de la participation à la fois des développeurs et des utilisateurs. Annoncer, c'est comme planter une graine. La nouvelle peut mettre du temps à se répandre. Si le projet récompense systématiquement ceux qui s'investissent, la nouvelle se répandra malgré tout, parce que les gens parlent quand ils ont bien aimé quelque chose. Si tout se passe bien, les lois exponentielles des réseaux de communications transformeront doucement votre projet en une communauté complexe, où vous ne connaîtrez pas forcément chacun par son nom, et où vous ne pourrez plus suivre toutes les discussions. Les prochains chapitres traiteront du travail dans cet environnement.

Infrastructure technique

Les technologies des logiciels libres reposent sur l'acquisition et l'intégration sélectives des informations. Plus vous serez habile à employer ces technologies et à persuader les autres de les employer, plus votre projet aura de succès. Cela devient d'autant plus vrai quand votre projet prend de l'ampleur. La bonne gestion de l'information empêche des projets Open Source de s'effondrer sous l'effet de la loi [1] de Brooks [2] [3] selon laquelle ajouter de la main-d'œuvre à un projet informatique en retard ne fait que le retarder plus encore. Fred Brooks a observé que la complexité d'un projet informatique augmente proportionnellement au carré du nombre de participants. Quand quelques personnes seulement sont impliquées, la communication est aisée, mais quand ce sont des centaines, comment savoir ce que chacun fait ? Pour bien gérer un projet de logiciel libre, il faut faire en sorte que chacun ait l'impression de travailler avec tous les autres dans la même pièce, mais une question s'impose : que se passe-t-il dans une pièce surpeuplée, quand tout le monde veut parler en même temps ?

Ce problème n'est pas nouveau. Dans les salles surpeuplées, la solution est le procédé parlementaire : des règles formelles sur les moyens de mener des discussions en temps réel dans de larges assemblées, pour s'assurer que d'importantes divergences d'opinion ne vont pas se retrouver noyées sous des flots de « moi, je », pour former des sous-comités, pour identifier quand des décisions sont prises, etc. L'un des rôles importants de ce processus est de spécifier comment le groupe interagit avec son système d'information. Certaines remarques ont vocation d'être « enregistrées », d'autres non. Cet « enregistrement » lui-même est sujet à manipulation et n'est pas une transcription littérale de ce qui vient de se produire, mais une

1. http ://fr.wikipedia.org/wiki/Loi_de_Brooks
2. http ://fr.wikipedia.org/wiki/Le_Mythe_du_mois-homme
3. Tiré de son livre, Frederick P. Brooks, *Le Mythe du mois-homme*, Paris, Vuibert, 1996 (1975).

représentation de ce que le groupe a convenu de considérer comme résultat. Pas d' « enregistrement » monolithique, mais à géométrie variable selon la finalité recherchée. Il comprend la totalité des comptes-rendus de toutes les réunions, les résumés, les ordres du jour et leurs annotations, les rapports de comité, les rapports des correspondants non présents, les listes d'actions à entreprendre, etc.

Comme Internet n'est pas vraiment une salle, la stratégie parlementaire qui fait taire les uns pendant que les autres parlent ne nous concerne pas. Mais quand on en vient aux techniques de gestion de l'information, on voit que les projets Open Source bien dirigés utilisent ce procédé parlementaire boosté aux stéroïdes. Puisque presque toute la communication dans les projets Open Source se fait par écrit, des systèmes élaborés ont évolué pour permettre l'étiquetage, le stockage, la recherche et le cheminement des données de façon appropriée, pour minimiser les répétitions afin d'éviter des divergences inutiles, pour corriger les informations fausses ou désuètes et pour connecter des bribes d'informations disparates au fur et à mesure que de nouveaux liens se créent. Les participants actifs dans les projets Open Source ont adopté plusieurs de ces techniques et réalisent souvent de complexes opérations manuelles pour s'assurer que l'information est correctement transmise. Mais le plus gros de l'effort repose finalement sur la sophistication du logiciel sous-jacent. Les médias de communication eux-mêmes devraient assurer l'acheminement autant que faire se peut, l'étiquetage et l'enregistrement devraient mettre l'information à disposition d'une façon la plus commode possible. Dans la pratique, évidemment, l'intervention humaine reste nécessaire à de nombreuses étapes du processus, et il est important que le logiciel la rende aussi aisée que possible. En règle générale, si l'humain fait bien attention à l'étiquetage et à l'acheminement de l'information en entrée du système, le logiciel doit être configuré pour utiliser au mieux toutes ces métadonnées.

Quant à la gestion de l'information, il n'existe pas de recette miracle car les paramètres sont nombreux. Vous pouvez très bien être parvenu à tout configurer de manière optimale selon vos besoins et avoir une communauté active, il arrivera sûrement un moment où la croissance du projet rendra certaines de ces pratiques inadaptées. Il se peut aussi que la croissance du projet se stabilise et que les communautés de développeurs et d'utilisateurs s'installent dans une relation satisfaisante avec l'infrastructure technique, puis que quelqu'un vienne, invente un nouveau service de gestion de l'information, et l'on vous demandera aussitôt pourquoi votre projet ne l'emploie pas. C'est, par exemple, ce qui arrive en ce moment à un certain nombre de projets Open Source antérieurs à l'invention du Wiki [1]. Beaucoup d'interrogations relèvent surtout d'une question de point de vue, comme la différence entre l'ergonomie pour ceux qui produisent l'information et l'ergonomie pour ceux qui la consomment, ou entre le temps requis pour configurer un logiciel de gestion de l'information et les avantages qu'il peut apporter au projet.

Prenez garde à la tentation d'automatiser les choses qui exigent vraiment une attention humaine. L'infrastructure technique est importante, mais ce qui fait fonctionner un projet Open Source, c'est l'attention et l'expression intelligente de cette attention que les humains impliqués vont déployer. Le but principal de cette organisation est de fournir à l'utilisateur les instruments les plus adaptés pour agir.

1. http ://fr.wikipedia.org/wiki/Wiki

1. Les besoins d'un projet

La plupart des projets Open Source offrent un minimum d'outils pour la gestion de l'information :

Site Web — La vitrine de votre projet aux yeux du public (centralisé et à sens unique). Le site Web peut également servir d'interface administrative à d'autres outils du projet.

Listes de diffusion — Traditionnellement le principal moyen de communication, et aussi le plus actif, au sein du projet. C'est une bonne ressource pour garder trace des discussions.

Contrôle de versions — Permet aux développeurs de contrôler facilement les changements apportés au code, les régressions, et de gérer les branches de développements parallèles. Il permet à chacun d'observer les modifications du code.

Référencement de bogues — Permet aux développeurs d'avoir à disposition l'historique de leurs travaux, de se coordonner et de planifier les correctifs. Il Permet à chacun de connaître le statut précis des bogues, et les informations liées (par exemple, les conditions de leur reproductibilité). La même méthode peut d'ailleurs être employée pour faire le suivi, non seulement des bogues, mais également des tâches, des versions, des nouvelles fonctionnalités, etc.

Messagerie instantanée ou chat en temps réel — Un endroit pour les discussions et les échanges sur un mode de questions - réponses énoncées rapidement et simplement. N'est pas toujours archivé complètement.

Chacun de ces outils satisfait un besoin particulier, mais leurs fonctions sont étroitement liées et ils doivent être conçus pour fonctionner ensemble. Plus loin, nous verrons comment c'est possible, et surtout, comment les utiliser. Le site Web ne sera pas évoqué tout de suite, car il s'agit plus d'un ciment pour les autres composants que d'un outil à part entière.

Vous pouvez éviter les prises de tête liées à leur choix et à leur configuration en optant pour une forge : un serveur qui offre, prêts à l'emploi, des modèles avec tous les outils nécessaires pour gérer un projet Open Source. Voir la section appelée « Les forges » plus loin dans ce chapitre pour une évaluation des avantages et des inconvénients de ce système.

2. Les listes de diffusion

Les listes de diffusion sont la base de la communication au sein d'un projet. Si un utilisateur rencontre un espace de dialogue, en dehors des pages Web, il y a de fortes chances que ce soit une des listes de diffusion du projet. Mais avant d'expérimenter la liste de diffusion elle-même, il sera en contact avec l'interface de la liste de diffusion ; c'est-à-dire le mécanisme par lequel il peut rejoindre la liste (« s'abonner »). Ceci nous amène à la règle n°1 des listes de diffusion :

> *« N'essayez pas de gérer une liste de diffusion à la main : procurez-vous un logiciel de gestion de listes. »*

Il serait tentant de repousser cela à plus tard. Le temps passé à installer un logiciel de gestion de listes peut sembler peu rentable au début. Gérer à la main de petites listes générant peu de trafic semble séduisant : établissez simplement une adresse d'abonnement qui redirige vers votre boîte mail, et quand quelqu'un l'utilise, ajoutez (ou enlevez) son adresse mail dans un fichier texte qui contient toutes les adresses de la liste. Qu'y a-t-il de plus simple ?

Le hic, c'est qu'une bonne gestion de listes de diffusion, ce que les gens sont en droit d'attendre, n'est pas simple du tout. Il ne s'agit pas simplement d'abonner et de désabonner les utilisateurs quand ils le demandent. Il s'agit également de faire de la modération pour empêcher le spam, d'offrir des versions résumées, et, message par message, de fournir de l'information standard et de l'information orientée projet grâce à des messages pré-écrits, ainsi que diverses autres choses. Un être humain gérant lui-même une adresse d'abonnement ne peut assurer que le strict minimum et n'est pas aussi fiable et performant qu'un logiciel.

Les logiciels modernes de gestion de liste offrent au minimum les fonctionnalités suivantes :

Inscription par e-mail et par le web — Quand un utilisateur s'abonne à une liste, il devrait recevoir, en réponse automatique par retour, un message d'accueil lui indiquant ce à quoi il s'est abonné, quels sont les possibilités offertes par le logiciel de liste de diffusion, et (ce qui est le plus important) comment se désinscrire. Bien sûr, cette réponse automatique peut être personnalisée pour donner plus d'informations sur le projet, comme par exemple son adresse, où trouver la FAQ, etc.

L'abonnement en mode résumé ou message par message — En mode résumé, l'abonné reçoit un courrier par jour contenant toutes les activités du jour de la liste. Pour qui suit une liste de manière détachée, sans participer, le mode résumé est souvent préférable car il permet de survoler rapidement tous les sujets et évite la distraction de recevoir des e-mails à n'importe quel moment.

Les possibilités de modération — La modération sert à vérifier les messages pour s'assurer qu'il ne s'agit ni de spam ni d'envois hors sujet avant que d'être envoyés à la liste entière. La modération demande une intervention humaine, mais les logiciels peuvent mâcher une grosse partie du travail. Nous y reviendrons.

Interface d'administration — L'interface d' administration permet à l'administrateur, entre autres choses, de retirer facilement les adresses obsolètes. Cela peut devenir urgent quand l'adresse d'un destinataire commence à renvoyer automatiquement des messages du type « Je ne suis plus à cette adresse » à toute la liste pour chaque e-mail reçu (certains logiciels de listes de diffusion peuvent même les détecter seuls et désabonner ces personnes automatiquement).

Manipulation des en-têtes — Beaucoup de gens mettent en place des filtrages sophistiqués et des règles de réponse dans leur logiciel de messagerie. Les logiciels de listes de diffusion peuvent ajouter et manipuler certains en-têtes standards pour permettre à ces personnes d'en tirer parti (nous y reviendrons).

Archivage — Tous les messages des listes sont enregistrés et mis à disposition sur le Web, certains logiciels de listes de diffusion proposent des interfaces spéciales pour assurer leur compatibilité avec des utilitaires d'archivage tiers comme MHonArc [1].

1. http ://www.mhonarc.org/

> Comme nous le verrons dans la section « Utilisation visible des archives » du le chapitre 6, l'archivage est crucial.

Retenez simplement ici que la gestion des listes de diffusion est un problème complexe, ayant déjà reçu beaucoup d'attention, mais en grande partie résolu. Vous n'êtes pas obligé de devenir expert sur le sujet, mais vous devriez savoir qu'il y a toujours de nouvelles choses à découvrir et que cette gestion demandera votre attention de temps à autre au cours de la vie de votre projet de logiciel libre. Nous allons désormais examiner quelques-uns des principaux problèmes rencontrés lors de la configuration des listes de diffusion.

Se prémunir du spam

Entre le moment où j'écris cette phrase et le moment où elle sera publiée, le problème du spam sur Internet aura sûrement pris des proportions beaucoup plus importantes ou, en tout cas, sera ressenti comme tel. Il fut un temps, il n'y a pas si longtemps, où l'on pouvait créer une liste de diffusion sans avoir à prendre de mesures de protection contre le spam. De temps en temps, on pouvait recevoir un e-mail égaré, mais c'était suffisamment rare pour que cela reste peu gênant. Cette âge d'or est révolu. De nos jours, une liste de diffusion qui ne se prémunit pas du spam sera rapidement noyée sous les e-mails indésirables, au point d'en devenir inutilisable. Les protections contre le spam sont indispensables. On peut séparer les protections contre le spam en deux catégories : celles qui empêchent les courriers indésirables d'apparaître sur la liste de diffusion et celles qui protègent les listes de diffusion contre les collecteurs d'adresses des spammeurs. La première étant la plus importante, c'est celle que nous allons détailler en premier.

Filtrer les messages

Il existe trois techniques de base pour éviter les messages indésirables, la plupart des logiciels de listes de diffusions les proposent toutes les trois. Il vaut mieux les utiliser de concert :

1. Autoriser automatiquement les messages uniquement envoyés par les abonnés

Cette méthode remplit très bien son rôle et ne demande que peu de travail : en général, il suffit de modifier un paramètre dans les réglages du logiciel de liste de diffusion. Mais prenez garde, les messages qui ne sont pas automatiquement approuvés ne doivent pas être rejetés pour autant ; ils devraient subir une inspection pour deux raisons : d'abord, vous feriez mieux de laisser la possibilité aux non-abonnés d'envoyer des messages (une personne qui a une question, ou une idée à soumettre, ne devrait pas avoir besoin de s'inscrire à la liste de diffusion juste pour y envoyer un message), ensuite, même les abonnés envoient parfois des messages depuis d'autres adresses que celle qu'ils ont utilisée pour s'inscrire. L'adresse e-mail n'est pas une méthode sûre pour identifier les personnes, et par conséquent ne doit pas servir à cela.

2. Filtrer les messages grâce à un logiciel de filtrage

Si la liste de diffusion le permet (la plupart le font), vous pouvez filtrer les messages grâce à un logiciel de filtrage de spam. Le filtrage automatique des spams n'est pas parfait (et ne

le sera jamais) vu que les spammeurs et les développeurs de filtres se sont engagés dans une course à l'armement sans fin. Malgré cela, le filtre peut largement réduire le nombre de spams en attente de modération. Comme la longueur de la liste d'attente se traduit en temps de travail manuel, tout gain obtenu à ce niveau, grâce au filtrage automatique, est bon à prendre.

Je ne peux pas détailler ici la mise en place des filtres à spam. Je vous renvoie donc à la documentation de votre logiciel de liste de diffusion pour en savoir plus (voir la section appelée « Les logiciels » plus loin dans ce chapitre). Les logiciels de liste de diffusion incluent souvent des fonctionnalités anti-spam, mais vous pouvez aussi choisir d'utiliser un programme de filtrage tiers. J'apprécie les programmes SpamAssassin [1] et SpamProbe [2] que j'ai utilisés avec une grande satisfaction. Je ne ferai pas de liste exhaustive, il existe bien d'autres logiciels Open Source de filtrage de spam, dont certains semblent très performants.

3. Modération

En ce qui concerne les courriels qui ne sont pas automatiquement admis parce qu'ils n'émanent pas d'un abonné, qui passent au crible d'un éventuel logiciel anti-spam, la dernière étape est la modération : le mail est redirigé vers une adresse spéciale où une personne l'examinera et l'acceptera ou non.

Accepter un message peut se faire de deux manières différentes : vous pouvez autoriser le message juste cette fois, ou encore, dire au logiciel de liste de diffusion de laisser passer dans le futur tous les messages de cet expéditeur. En général, c'est la deuxième option qui est favorisée afin de faciliter la tâche de modération à l'avenir. La manière de procéder est différente selon les systèmes, mais, en principe il faut répondre à une adresse particulière en incluant la commande « accepter » (ce qui signifie « accepter uniquement ce message ») ou « autoriser » (autoriser ce message ainsi que tous les futurs messages).

Le rejet se fait simplement en ignorant le courrier de modération. Si le logiciel de la liste de diffusion ne reçoit jamais de consigne pour dire qu'un message est valide, alors, il ne fera pas suivre ce message sur la liste : laisser le message de côté aura donc l'effet désiré. Il arrivera parfois que vous ayez la possibilité de répondre avec une commande « rejeter » ou « empêcher » pour rejeter automatiquement et de façon permanente les messages de cet utilisateur sans même qu'ils ne repassent par la case « Modération ». En général, ce n'est pas très utile puisque la modération sert principalement à éviter le spam, et que les spammeurs utilisent rarement la même adresse deux fois.

La modération doit servir uniquement au filtrage des spams et des messages hors sujet, ou envoyés sur la mauvaise liste de diffusion. Le système de modération devrait vous fournir un moyen de répondre directement à l'expéditeur, mais n'employez pas cette méthode pour répondre directement à une question adressée à la liste de diffusion, même si vous pouvez fournir une réponse rapidement. Fonctionner ainsi empêcherait le projet de se faire une idée précise du genre de questions que les gens se posent et enlèverait aux membres l'occasion de répondre aux questions eux-mêmes, ou de voir les réponses des autres. La modération des listes de diffusion doit se borner à l'entretien de la liste de diffusion, rien d'autre.

1. http ://spamassassin.apache.org/
2. http ://spamprobe.sourceforge.net/

Masquer les adresses dans les archives

Pour éviter que vos listes de diffusion ne deviennent une mine d'adresses pour les spammeurs, une technique courante est de masquer les adresses e-mail des personnes dans les archives en remplaçant par exemple

`a.nonyme@undomaine.com`

par

`a.nonyme_AT_undomaine.com`

ou par

`a.nonymeNOSPAM@undomaine.com`

ou d'autres codes similaires évidents pour un humain. Comme les collecteurs d'adresses à spammer fonctionnent souvent en naviguant sur les pages Web (y compris dans vos archives de listes de diffusion) à la recherche de séquences contenant @, coder les adresses e-mail est une manière de les rendre invisibles ou inutilisables par les spammeurs. Cela ne change rien à la quantité de spam envoyée directement à la liste de diffusion évidemment, mais au moins vous évitez d'en augmenter le nombre.

Le masquage d'adresse peut être sujet à controverse. Certaines personnes appréciant beaucoup cette technique seront surprises si vos archives ne le font pas automatiquement. D'autres pensent plutôt que c'est un désagrément (parce que les utilisateurs doivent aussi traduire les adresses avant usage). Certains doutent de l'efficacité de la méthode puisqu'un collecteur peut, en théorie, s'adapter aux codes les plus répandus. Notez, malgré tout, que, par expérience, le masquage d'adresse se montre efficace [1].

Idéalement, le programme de gestion de listes devrait laisser le choix à chaque abonné, grâce à un en-tête oui/non ou un paramètre dans les préférences de son compte. Mais je ne connais aucun logiciel permettant ce réglage, ce qui oblige, pour l'instant, le responsable des listes à faire ce choix pour tout le monde (en supposant que le logiciel d'archivage propose cette option, ce qui n'est pas toujours le cas). Je penche légèrement en faveur du masquage d'adresses. Certaines personnes sont très prudentes et n'affichent pas leur adresse e-mail sur les pages Web ou à n'importe quel endroit qu'un collecteur d'adresse pourrait inspecter. Elles seraient déçues de voir tous leurs efforts réduits à néant par une archive de liste de diffusion. De plus, le désagrément imposé aux utilisateurs des archives par le masquage est très faible puisqu'il est fort simple de « traduire » ces adresses si vous avez besoin de contacter quelqu'un. Mais n'oubliez pas qu'au final, cela reste une course à l'armement : au moment où vous lirez ceci, les collecteurs auront peut-être évolué au point qu'ils pourront reconnaître les techniques classiques de masquage d'adresse e-mail et nous devrons trouver alors une autre parade.

Identification et gestion des en-têtes

Les utilisateurs des listes rangeront souvent les messages dans des dossiers réservés au projet, séparés de leurs autres courriers. Leur logiciel de lecture de courrier peut faire cela

1. http ://www.cdt.org/speech/spam/030319spamreport.shtml

automatiquement en vérifiant les en-têtes du message. Les en-têtes sont les champs au début du courrier qui indiquent l'expéditeur, le destinataire, le sujet, la date et d'autres informations à propos du message. Certains en-têtes sont bien connus et obligatoires :

De : ...
À : ...
Sujet : ...
Date : ...

D'autres sont optionnels bien que plutôt courants. Par exemple, vous n'êtes pas obligés de remplir l'en-tête :

```
Répondre à: expediteur@adresse.courriel.ici
```

Mais la plupart des gens le font puisque cela permet au destinataire de répondre de manière certaine à l'auteur du message (c'est particulièrement utile si l'auteur a dû recourir à une adresse différente de celle à laquelle les réponses devraient être adressées).

Certains logiciels de courrier fournissent une interface simple d'emploi pour trier les messages en fonction du sujet. Certaines personnes demandent par conséquent que les listes de diffusion ajoutent automatiquement un préfixe aux sujets afin que leur logiciel puisse automatiquement ranger ces messages dans le bon dossier. L'idée est que l'auteur du message écrive :

```
Sujet: Préparation de la version 2.5
```

mais que le message au final soit envoyé sous cette forme (par exemple) :

```
Sujet: [discussion@listes.exemple.org] Préparation de la
version 2.5
```

Bien que la plupart des logiciels de gestion de listes de diffusion proposent cette option, je ne vous recommande pas de l'activer. Le problème réglé ici peut l'être par des moyens beaucoup moins marqués, et le prix à payer, l'utilisation de l'espace dans le champ *Sujet*, est bien trop élevé. Les utilisateurs habitués aux listes de diffusion passent en général en revue les sujets des messages pour décider de ce qu'ils vont lire et ce à quoi ils vont répondre. Ajouter le nom de la liste au sujet peut repousser la partie importante du sujet hors de l'écran, la rendant ainsi invisible. Cela masque les informations sur lesquelles les gens se reposent durant leur inspection des sujets, réduisant par conséquent l'utilité de la liste de diffusion. Plutôt que de grignoter une partie du champ *Sujet*, enseignez aux utilisateurs à utiliser les autres champs, en commençant par le champ *À :* qui devrait afficher le nom de la liste :

```
À: <discussion@listes.exemple.org>
```

N'importe quel logiciel de messagerie pouvant filtrer les sujets devrait également pouvoir filtrer aussi facilement le champ *À :*. D'autres champs optionnels sont en général remplis

pour les listes de diffusion. Baser le filtrage sur ces champs est encore plus efficace que d'utiliser les champs *À :* ou *Cc :* puisque ces champs sont remplis automatiquement par le logiciel de la liste de diffusion, certains utilisateurs s'attendent à les trouver :

```
  list-help: mailto:discuss-help@lists.example.org

  list-unsubscribe: mailto:discuss-unsubscribe@lists.example.org

  list-post: mailto:discuss@lists.example.org

  Delivered-To: mailing list discuss@lists.example.org

  Mailing-List: contact discuss-help@lists.example.org; run
by ezmlm
```

Pour la plupart, leur fonction est évidente. Voyez nisto.com [1] pour plus d'informations à ce sujet ou, si vous cherchez des spécifications vraiment détaillées et formelles, voyez faqs.org [2].

Vous remarquerez que ces champs suggèrent que si vous avez une liste nommée *list*, alors vous possédez aussi les listes administratives *list-help* et *list-unsubscribe*. En plus de celles-ci, il est normal que vous proposiez *list-subscribe* pour s'inscrire et *list-owner* pour contacter les administrateurs des listes. En fonction du logiciel utilisé pour gérer vos listes, ces adresses et/ou plusieurs autres adresses administratives peuvent être mises en place, vous trouverez des explications dans la documentation. En général, un descriptif complet de toutes ces adresses spéciales est communiqué à chaque nouvel utilisateur au sein d'un message de bienvenue lors de l'inscription. Vous recevrez aussi certainement une copie de ce courrier. Si vous ne la recevez pas, alors demandez ce que reçoivent les utilisateurs lorsqu'ils s'inscrivent à une liste. Gardez ce message à portée afin de pouvoir répondre aux questions concernant les fonctions de la liste de diffusion ou, encore mieux, affichez-le sur une page Web, ainsi quand quelqu'un perd sa copie des instructions et envoie un message concernant les modalités de désinscription, vous n'avez qu'à lui envoyer l'URL.

Certains logiciels de liste de diffusion possèdent une option pour ajouter les instructions de désabonnement à chaque message. Si cette option est présente, activez-la. Cela n'ajoute que quelques lignes au message, ne gêne pas sa lecture, et peut vous faire gagner beaucoup de temps en évitant que les gens ne vous écrivent, ou pire encore, n'écrivent à toute la liste, pour demander comment se désinscrire.

Le grand débat du « Répondre à »

Précédemment, dans une partie appelée « Éviter les discussions privées », j'ai insisté sur l'importance du fait que les discussions doivent se dérouler dans les forums publics, et j'ai dit que parfois des mesures devaient être prises pour éviter que les conversations ne se transforment en échanges d'e-mails privés. Ce chapitre traite de la mise en place des logiciels de communication du projet afin qu'ils facilitent au maximum la vie du projet. Par conséquent, si le logiciel de gestion des listes de discussion vous offre la possibilité de garder automatiquement les discussions sur la liste, vous vous direz qu'il est logique d'activer cette

1. http ://www.nisto.com/listspec/list-manager-intro.html
2. http ://www.faqs.org/rfcs/rfc2369.html

fonctionnalité. En fait ce n'est pas si évident. Cette option existe, mais elle comporte des inconvénients plutôt restrictifs. Son utilisation est sujette à l'un des plus importants débats concernant la gestion des listes de diffusion, rien qui ne fera la une des journaux du soir, mais parfois les discussions à ce propos peuvent devenir tendues au sein des projets de logiciels libres. Ci-dessous je vais décrire la fonctionnalité, exposer les principaux arguments de chaque camp et vous donner mes meilleures recommandations.

La fonctionnalité en elle-même est très simple : le logiciel peut, si vous le souhaitez, remplir le champ *Répondre à :* automatiquement afin que les réponses soient redirigées sur la liste de diffusion. C'est à dire que, peu importe ce que l'expéditeur met dans le champ *Répondre à :* (ou même s'il ne le remplit pas), quand les abonnés de la liste recevront le message l'en-tête contiendra l'adresse de la liste :

```
Répondre à: discuss@lists.example.org
```

De prime abord, cela semble être une bonne chose parce que quasiment tous les logiciels de messagerie inspectent le champ *Répondre à :* et donc quand quelqu'un enverra une réponse, elle sera automatiquement envoyée à la liste entière : pas uniquement à l'expéditeur du message auquel on répond. Bien sûr, la personne qui répond peut modifier à la main le destinataire du message, mais l'important est que, par défaut, les réponses sont directement envoyées à la liste. C'est un très bon exemple d'utilisation de la technologie pour encourager la collaboration.

Malheureusement, il existe quelques inconvénients. Le premier est connu sous le nom du problème de « Je ne peux plus retrouver mon chemin » : il se peut que l'expéditeur mette sa « véritable » adresse e-mail dans le champ *Répondre à :* parce que, pour une raison ou pour une autre, il a utilisé une autre adresse pour envoyer le message que celle qu'il utilise pour les recevoir. Les personnes qui expédient et lisent les messages à partir de la même adresse n'ont pas ce problème, et sont même parfois surprises d'apprendre son existence. Mais, pour ceux qui utilisent leurs comptes mail de manière particulière ou qui n'ont pas de contrôle sur le champ *De :* dans leurs courriers (parce qu'ils écrivent depuis leur travail, ou parce qu'ils n'ont pas assez d'influence sur le département informatique), utiliser le champ *Répondre à :* peut être la seule manière d'être sûr que les réponses leur parviennent. Quand une personne dans cette situation envoie un message à une liste de diffusion à laquelle il n'est pas abonné, l'adresse dans le champ *Répondre à :* devient une information essentielle. Si le logiciel remplace cette adresse, il se peut qu'il ne reçoive jamais de réponse.

Le deuxième inconvénient est lié aux attentes et, d'après moi, c'est l'argument qui a le plus de poids contre l'automatisation du champ *Répondre à :*. La majorité des gens ayant l'habitude de se servir des e-mails sont accoutumés à deux choix simples pour répondre : Répondre à tous et Répondre. Tous les logiciels modernes de messagerie possèdent deux boutons distincts pour ces deux fonctions. Les utilisateurs savent que pour répondre à tout le monde (ce qui inclut la liste), ils doivent utiliser le bouton « Répondre à tous », et que pour répondre en privé à l'auteur ils doivent utiliser le bouton « Répondre ». Même si votre but est d'encourager les gens à répondre à toute la liste le plus souvent possible, il y aura parfois des circonstances qui font qu'une réponse privée est plus appropriée, par exemple, si une personne veut dire quelque chose de confidentiel à l'auteur du message original, quelque chose qui n'aurait pas sa place sur la liste publique.

Maintenant penchons-nous sur le cas où la liste a ré-écrit le champ *Répondre à :*. La personne qui répond appuie sur le bouton « Répondre » en s'attendant à envoyer un message privé à l'auteur du courrier. Puisque c'est ce qu'il se passe normalement, il ne prendra pas forcément la peine de vérifier l'adresse du destinataire du message. Il écrit alors son message privé et confidentiel, où il pourrait dire des choses gênantes sur une autre personne de la liste, et appuie ensuite sur « Envoi ». Alors qu'il ne s'y attendait pas, quelques minutes plus tard son message apparaît sur la liste de diffusion ! Il aurait effectivement, en théorie, dû prendre le temps de regarder avec précaution le champ *Destinataire*, et n'aurait pas dû supposer qu'il n'avait pas à se soucier du champ *Répondre à :*. Mais les expéditeurs règlent quasiment à chaque fois le champ *Répondre à :* de telle sorte qu'ils reçoivent la réponse (ou pour être plus précis, c'est leur logiciel de messagerie qui le fait pour eux), et beaucoup d'utilisateurs expérimentés le prennent pour argent comptant. En fait, lorsqu'une personne met une autre adresse que celle de l'expéditeur dans le champ *Répondre à :*, comme celle de la liste par exemple, il prendra en général la peine de le notifier dans le message, ainsi les gens ne seront pas surpris par ce qui se passe lorsqu'ils appuient sur « Répondre ».

À cause des lourdes conséquences potentielles que cela peut entraîner, je préfère configurer le logiciel de gestion de liste de manière à ce qu'il ne modifie pas le champ *Répondre à :*. C'est l'un des cas où l'utilisation de la technologie pour encourager la collaboration peut, à mon sens, avoir des effets pervers. Mais l'autre camp a également d'excellents arguments à faire valoir. Quel que soit votre choix, des gens de temps à autre vous demanderont pourquoi vous n'avez pas fait l'autre choix. Comme c'est quelque chose que vous ne voulez pas voir prendre de trop grandes proportions, il vaut mieux que vous ayez une réponse toute prête, une réponse qui mettra un terme au débat plutôt que de l'encourager. Ne faites pas paraître votre décision, que vous choisissiez une solution ou l'autre, comme étant la seule, l'unique valable et la bonne (même si vous pensez que c'est le cas). Insistez plutôt sur le fait que c'est un très vieux débat, que les deux camps possèdent de bon arguments, mais qu'aucun choix ne peut satisfaire tous les utilisateurs, en conséquence, vous avez pris la décision qui vous semblait la meilleure. Demandez poliment à ce que le sujet ne soit pas ré-ouvert, à moins que quelqu'un ait quelque chose de vraiment nouveau à apporter au débat, puis ne participez plus à la discussion en espérant qu'elle s'éteigne d'elle-même. Quelqu'un pourrait suggérer de voter pour choisir une méthode ou l'autre. Vous pouvez le faire si c'est votre choix, mais je ne pense pas qu'un vote à main levée soit la meilleure solution dans ce cas. Le risque que quelqu'un se fasse surprendre par la modification du champ *Répondre à :* est trop important et le désagrément pour chacun est plutôt faible (devoir rappeler occasionnellement aux gens de répondre à la liste entière) pour qu'une majorité, même si c'est la majorité, impose un tel risque à une minorité. Je n'ai pas abordé, ici, tous les aspects de ce problème, seulement ceux qui semblaient les plus importants. Si le sujet vous intéresse je vous conseille de lire ces deux documents canoniques toujours cités dans ce débat :

- *Leave Reply-to alone*, de Chip Rosenthal ('Reply-To' Munging Considered Harmful. An Earnest Plea to Mailing List Administrators [1])
- *Set Reply-to to list*, de Simon Hill ('Reply-To' Munging Considered Useful An Earnest Plea to Mailing List Administrators [2])

1. http ://www.unicom.com/pw/reply-to-harmful.html
2. http ://www.metasystema.net/essays/reply-to.mhtml

Malgré cette légère préférence énoncée ci-dessus, je ne pense pas qu'il existe une vérité « transcendante » à sujet, et je participe gaiement à de nombreuses listes imposant le *Répondre à :*. Le mieux que vous puissiez faire, est d'opter assez tôt pour une solution ou pour l'autre, et d'éviter de vous faire attirer dans un débat par la suite.

Deux rêves

Un jour, quelqu'un aura cette idée brillante d'ajouter un bouton « Répondre à la liste » dans un logiciel de messagerie. Pour ce faire, il se servirait des en-têtes afin de déterminer l'adresse de la liste de diffusion, et enverrait donc la réponse directement à la liste, en ne se préoccupant pas des adresses d'autres destinataires puisqu'ils sont, de toute façon et pour la plupart déjà inscrits à la liste. Finalement, d'autres logiciels de messagerie reprendraient l'idée et le débat deviendrait obsolète (en fait, le logiciel de messagerie Mutt [1] propose déjà cette fonctionnalité. Il ne reste plus aux autres qu'à la copier [2]).

Une meilleure solution : laisser le choix à chaque abonné. Ceux qui veulent que la liste remplisse le champ *Répondre à :* (que se soit pour leurs propres messages ou ceux des autres) pourraient régler cette option et ceux qui ne le veulent pas pourraient aussi choisir. Je ne connais cependant aucun logiciel de gestion de liste qui permette ce choix individuel. Pour le moment on doit faire avec une configuration identique pour tous [3].

L'archivage

Les détails techniques concernant la mise en place de l'archivage d'une liste de diffusion sont particuliers au logiciel qui fait fonctionner la liste et dépassent le cadre de ce livre. Lors du choix, ou de la configuration, de l'archive, vous devez prendre en compte les facteurs suivants :

Mise à jour rapide

Les gens se réfèreront souvent à un message archivé envoyé une ou deux heures auparavant. Si possible, le logiciel devrait archiver chaque message instantanément, dès qu'un message apparaît dans la liste de diffusion, afin qu'il soit aussi présent dans les archives. Si cette option n'est pas disponible, alors essayez de faire en sorte que l'archive soit remise à jour au moins toutes les heures (par défaut certains logiciels d'archives se mettent à jour automatiquement chaque nuit, mais dans la pratique ce délai est bien trop important pour une liste de diffusion active).

La stabilité du référentiel

1. http ://www.mutt.org/

2. Peu après la publication de ce livre, Michael Bernstein m'a écrit pour m'informer que « Mutt n'est pas le seul logiciel de messagerie à proposer la fonction « Répondre à la liste ». Par exemple Evolution le propose grâce à un raccourci clavier, mais pas avec un bouton (Ctrl+L). »

3. Depuis la rédaction de ce paragraphe, j'ai appris qu'il existe au moins un système de gestion de liste qui propose cette fonction : Siesta. Je vous encourage à lire l'article de Simon Wistow sur perl.com pour en savoir plus.

Une fois qu'un message est archivé à une URL donnée, il devrait rester accessible par la même URL à tout jamais, ou le plus longtemps possible en tout cas. Même si les archives sont reconstruites, rétablies à partir d'une sauvegarde ou réparées d'une quelconque manière, aucune URL qui a été rendue publique ne devrait être modifiée. Des coordonnées stables rendent possible l'indexation des archives par les moteurs de recherche qui sont les meilleurs compagnons des utilisateurs en quête de réponses. Des coordonnées stables sont aussi importantes, car les messages et les sujets, dans les listes de diffusion, contiennent souvent des liens vers le système de suivi de bogues (voir la section nommée « Système de suivi de bogues » plus loin dans ce chapitre) ou vers d'autres documents du projets.

Idéalement, les logiciels de listes de diffusion devraient inclure l'URL du message dans les archives, ou, au moins, une portion particulière de l'URL dans les en-têtes du message lorsqu'il est distribué aux destinataires. Ainsi, ceux qui ont une copie du message peuvent savoir où il est rangé dans les archives sans avoir à se rendre sur la page des archives. C'est utile en effet, car toute opération nécessitant l'utilisation d'un navigateur prend du temps. Par contre, je ne sais pas s'il existe un logiciel proposant cette fonctionnalité. Ceux que j'ai utilisés ne le faisaient pas. C'est tout de même une fonctionnalité que vous devriez chercher (ou, si vous écrivez un logiciel de gestion de listes de diffusion, c'est une fonctionnalité que vous devriez penser à ajouter, s'il vous plaît).

Les sauvegardes

La méthode de sauvegarde des archives devrait être plutôt évidente, et la manière de les restaurer ne devrait pas être trop compliquée non plus. En d'autres termes, ne voyez pas votre logiciel d'archivage comme une boîte noire. Tout le monde (vous ou quelqu'un de votre projet) devrait savoir où les messages sont classés, et comment recréer, si nécessaire, la page d'archives depuis les messages sauvegardés. Les archives sont des données précieuses ; un projet qui les perd voit disparaître une grande partie de sa mémoire collective.

La gestion des sujets

Il devrait être possible de naviguer depuis n'importe quel message vers le sujet (groupe de messages ayant un lien) auquel appartient ce message. Chaque sujet devrait avoir sa propre URL, différente de l'URL des messages composant ce sujet.

Recherche

Un logiciel d'archivage ne proposant pas d'outil de recherche, ni dans le corps des messages, ni par auteur ou par sujet, est quasiment inutile. Remarquez que certains logiciels proposent un outil de recherche simplement en sous-traitant le travail à un moteur de recherche tiers comme Google. Passons. Mais un outil de recherche natif est en général plus précis puisqu'il permet à l'utilisateur de spécifier, par exemple, que le mot doit se trouver dans le sujet et pas dans le corps du texte.

Ceci n'est qu'une liste technique pour vous aider à évaluer et mettre en place un outil d'archivage. Comment amener les gens à vraiment utiliser cet outil pour le bien du projet, est un sujet différent, qui sera abordé dans d'autres chapitres de cet ouvrage, en particulier dans la section intitulée « Utilisation visible des archives ».

Les logiciels

Voici une liste d'outils Open Source pour la gestion et l'archivage de listes. Si le site qui héberge votre projet propose déjà une configuration par défaut, vous n'aurez peut-être jamais à faire de choix. Mais si vous devez en installer un vous-même, en voici quelques-uns. Ceux que j'ai utilisés sont Mailman, Ezmlm, MHonArc et Hypermail, mais cela ne veut pas dire que les autres ne sont pas bons (et bien sûr, il en existe probablement sur lesquels je ne suis pas tombé ; cette liste n'est en rien exhaustive). Les logiciels de gestion de listes de diffusion sont :

- **Mailman** : http ://www.list.org/ (Possède un outil d'archivage inclus, et s'adapte bien aux outils d'archivage tiers.)
- **SmartList** : http ://www.procmail.org/ (Fait pour être utilisé avec le système d'envoi de courrier Procmail.)
- **Ecartis** : http ://www.ecartis.org/
- **ListProc** : http ://listproc.sourceforge.net/
- **Ezmlm** : http ://cr.yp.to/ezmlm.html (Fait pour être utilisé avec le système d'envoi de courrier Qmail.)
- **Dada** : http ://mojo.skazat.com/ (Malgré les tentatives étranges faites sur le site Web pour s'en cacher, c'est bien un logiciel libre, publié sous licence GNU General Public License. Il propose aussi un outil d'archivage incorporé.)

Les logiciels d'archivage sont :

- **MHonArc** : http ://www.mhonarc.org/
- **Hypermail** : http ://www.hypermail.org/
- **Lurker** : http ://sourceforge.net/projects/lurker/
- **Procmail** : http ://www.procmail.org/ (Logiciel fonctionnant de pair avec SmartList, c'est un système d'envoi de courrier qui peut apparemment être configuré pour réaliser l'archivage également.)

3. Les logiciels de gestion de versions

Un logiciel de gestion de versions (ou *logiciel de gestion de révisions*) est un mélange de technologie et de bonnes pratiques pour traquer et contrôler les modifications apportées aux fichiers d'un projet, en particulier au code source, à la documentation et aux pages Web. Si vous n'avez jamais utilisé un logiciel de gestion de version, la première chose que vous devriez faire est de trouver qui en a l'expérience et la maîtrise, et le convaincre de rejoindre le projet. De nos jours, tout le monde s'attend au minimum à ce que le code source du projet soit sous la surveillance d'un logiciel de gestion de versions, et votre projet ne sera pas pris au sérieux s'il n'utilise pas efficacement un tel logiciel.

Les logiciels de gestion de versions sont devenus des standards, car ils fournissent une aide précieuse dans quasiment chaque domaine d'un projet efficace : la communication entre développeurs, la gestion des sorties, la gestion des bogues, la stabilité du code, le développement expérimental, les attributions et les autorisations de modifications. La gestion de versions vous fournit un contrôle centralisé sur tous ces domaines. Le cœur de la gestion

de versions est la gestion des modifications : l'identification de chaque petit changement apporté aux fichiers du projet, l'annotation de chaque modification par des métadonnées comme la date du changement, son auteur, et la possibilité de ressortir ces données pour toute demande, quelle qu'en soit la manière. C'est un mécanisme de communication avec lequel le changement est l'unité de base de l'information.

Nous n'aborderons pas tous les aspects de l'utilisation d'un logiciel de gestion de versions dans cette partie. La gestion de versions étant un vaste sujet, nous l'étudierons au fur et à mesure, tout au long du livre. Ici, nous allons nous intéresser plus particulièrement au choix et à l'installation d'un logiciel de gestion de versions, avec comme objectif la promotion du développement collaboratif.

Vocabulaire de la gestion de versions

Ce livre ne vous enseignera pas l'emploi de la gestion de versions si vous ne l'avez pas déjà expérimenté auparavant, cependant il serait impossible d'aborder ce sujet sans quelques termes clés. Ces termes sont utiles, indépendamment de tout système de gestion de versions : ce sont les noms et verbes de base de la collaboration en réseau, et ils seront employés de manière générique tout au long de ce livre. Même s'il n'existait aucun système de gestion de versions, le problème de gestion des modifications serait quand même présent, et ces mots nous fournissent un langage pour en parler de manière concise.

« Version » contre « Révision »
Le mot version est parfois employé comme synonyme de « révision ». Ici, je ne lui donnerai pas cette signification afin de ne pas le confondre trop facilement avec « version », dans le sens de version d'un logiciel, c'est à dire le numéro de sortie ou d'édition comme dans « Version 1.0 ». Mais l'expression « gestion de versions » étant déjà répandue, je continuerai à l'utiliser comme synonyme de « gestion de révisions » ou « gestion de modifications ».

Commit — Apporter une modification au projet, ou, plus formellement, enregistrer un changement dans la base de données de gestion de versions, pour qu'il puisse être ajouté dans une version future du projet. *Commit* peut être utilisé comme un verbe ou un nom. En tant que nom, il est surtout synonyme de « modification ». Par exemple : « Je viens juste d'enregistrer un correctif pour le bogue de crash de serveur que les gens ont rapporté sur Mac OS X. Jay, pourrais-tu, s'il te plaît, vérifier le *commit*, et t'assurer que je ne me trompe pas au sujet de l'allocation ? »

Messages enregistrés — Quelques commentaires joints à chaque *commit*, décrivant la nature et le but du *commit*. Les messages enregistrés font partie des documents les plus importants d'un projet : ils font le lien entre le langage très technique des modifications du code et le langage plus compréhensible qui se rapporte aux fonctionnalités, aux corrections de bogues et à la progression du projet. Par la suite, dans cette section, nous étudierons comment distribuer les messages enregistrés au bon public ; de plus, la section nommée « Codifier la tradition » dans le chapitre 6 aborde les manières d'encourager les participants à écrire des messages enregistrés concis et utiles.

Mise à jour — Demander que les autres modifications (*commit*) soient incorporées dans votre propre version du projet, c'est à dire, mettre votre copie à jour. C'est une opération de routine, la plupart des développeurs mettent à jour leur code plusieurs fois par jour. Ainsi, ils savent qu'ils utilisent sensiblement la même chose que les autres. En conséquence, s'ils détectent un bogue, il y a peu de chance qu'il ait déjà été corrigé. Par exemple : « Salut, j'ai remarqué que le code d'indexation oublie toujours le dernier nombre. Est-ce un nouveau bogue ? » « Oui, mais il a été réparé la semaine dernière, fais une mise à jour, il devrait disparaître. »

Dépôt — Une base de données au sein de laquelle les modifications sont stockées. Certains logiciels de gestion de versions sont centralisés : il y a un unique dépôt maître qui conserve toutes les modifications du projet. D'autres sont décentralisés : chaque développeur possède son propre dépôt et les modifications peuvent être partagées entre les dépôts de manière arbitraire. Le logiciel de gestion de versions conserve un suivi des dépendances entre les modifications. Au moment de la publication d'une nouvelle version, un ensemble particulier de modifications est approuvé pour la sortie. Quant à savoir quel système est le meilleur, centralisé ou décentralisé… Cette question est l'une des vieilles guerres du développement de logiciel, essayez de ne pas vous laisser entraîner dans ce débat sur l'une des listes du projet.

Retrait — L'obtention d'une copie du projet depuis le dépôt. Un retrait produit en général une arborescence de répertoires appelée « copie de travail » (voir ci-dessous), à partir de laquelle des changements peuvent être intégrés au dépôt originel. Pour certains logiciels décentralisés de gestion de versions, chaque copie de travail est, elle-même, un dépôt, et les modifications peuvent être envoyées (ou aspirées) vers les dépôts les acceptant.

Copie de travail — L'arborescence personnelle d'un développeur contient les fichiers du code source du projet, et peut également contenir pages Web et autres documents. Une copie de travail contient également quelques méta-données prises en charge par le logiciel de gestion de versions, indiquant à la copie de travail de quel dépôt elle provient, quelles « révisions » (voir ci-dessous) des fichiers sont présentes, etc. Généralement, chaque développeur possède sa propre copie de travail dans laquelle il réalise et teste les modifications, et à partir de laquelle il *commit*.

Révision, Modification et Ensemble de modifications — Une « révision » est en général une incarnation précise d'un fichier ou dossier particulier. Par exemple, si le projet commence avec la révision 6 du fichier F et qu'ensuite quelqu'un modifie F on parlera alors de la révision 7 de F. Certains systèmes parlent aussi de « révision », « modification » ou « ensemble de modifications » pour se référer à un ensemble de modifications ajoutées en même temps comme une unité conceptuelle. Ces termes ont parfois une signification technique distincte selon le logiciel de gestion de versions, mais l'idée générale est toujours la même : ils fournissent un moyen de parler sans ambiguïté d'un point précis dans l'histoire d'un fichier ou d'un ensemble de fichiers : par exemple, immédiatement avant ou après la correction d'un bogue, ou encore « Ah oui, elle a corrigé cela dans la révision 10 » ou bien « Elle a corrigé cela dans la révision 10 de foo.c. » Quand quelqu'un parle d'un fichier ou d'un ensemble de fichiers sans préciser de révision particulière, on comprend généralement qu'il s'agit de la révision la plus récente.

Diff — La représentation textuelle d'une modification. Un diff montre quelles lignes ont été modifiées et comment, en ajoutant quelques lignes de contexte d'un côté ou de l'autre. Pour un développeur déjà familier avec le code, la lecture d'un diff et du code suffisent, en général, à comprendre l'impact des modifications, voire à détecter des bogues.

Mot-clé — Une étiquette pour un ensemble de fichiers donnés à une révision donnée. Les mots-clés sont en général utilisés pour résumer les idées majeures du projet. Par exemple, un mot-clé est généralement utilisé pour chaque sortie publique afin qu'on puisse obtenir, directement depuis le logiciel de gestion de versions, l'ensemble exact des fichiers/révisions compris dans cette version. Des mots-clés courants sont Release-1-0, Delivery_00456, etc.

Branche — Une copie du projet, sous gestion de versions mais isolée, afin que les modifications de cette branche n'affectent pas le reste du projet (et vice versa) , sauf quand les modifications sont « fusionnées » volontairement dans un sens ou l'autre (voir plus bas). Les branches sont aussi connues sous le nom de « lignes de développement ». Même dans un projet n'ayant pas explicitement de branches, on considère toujours que le développement s'effectue sur la « branche principale », également connue sous le nom de « ligne principale » ou « tronc ». Les branches offrent la possibilité d'isoler différentes lignes de développement les unes des autres. Par exemple, une branche peut être employée pour faire du développement expérimental qui serait trop déstabilisant pour le tronc principal. Ou, à l'inverse, une branche peut être utilisée pour stabiliser une nouvelle version. Au cours du processus de sortie, le développement normal continue sans interruption dans la branche principale du dépôt, tandis que dans la branche de sortie aucun changement n'est accepté, sauf ceux approuvés par les responsables de la parution. Ainsi, la conception de la nouvelle version n'interfère pas avec le travail de développement en cours. Voir la section « Utilisez les branches pour éviter les embouteillages », plus loin dans ce chapitre pour une discussion plus détaillée à propos des branches.

Fusion (ou port) — Transférer une modification d'une branche à une autre. Cela englobe la fusion du tronc principal vers d'autres branches et inversement. En fait, ce sont les types de fusion les plus courants, il est rare de porter une modification entre deux branches secondaires. Voir la section appelée « Unicité de l'information » pour en savoir plus sur ce type de fusion. « Fusion » a un deuxième sens proche : c'est ce que fait le logiciel de gestion de versions quand il voit que deux personnes ont modifié le même fichier à des endroits différents. Puisque les deux modifications n'interfèrent pas entre elles, quand l'une des personnes met à jour sa copie du fichier (contenant déjà ses propres changements), les modifications de l'autre personne seront automatiquement fusionnées. C'est très courant, particulièrement dans les projets où plusieurs personnes travaillent sur le même code. Quand deux modifications différentes se chevauchent, il en résulte un « conflit », voir ci-dessous.

Conflit — C'est ce qui se passe quand deux personnes tentent de faire des changements au même endroit du code. Tous les systèmes de gestion de version détectent automatiquement les conflits, et avertissent au moins l'un des responsable de ces modifications conflictuelles. C'est alors à l'humain de régler le conflit, et d'envoyer la résolution au logiciel de gestion de version.

Verrouiller — Une manière de se réserver les modifications sur un fichier ou un dossier particulier. Par exemple : « Je ne peux pas envoyer de modifications des pages Web en ce moment. Il semblerait qu'Alfred les ait verrouillées pendant qu'il modifie leur image de fond. » Tous les systèmes de gestion de versions ne permettent pas ceci, et ceux qui l'autorisent n'imposent pas l'utilisation de cette fonctionnalité. On dit que les systèmes de gestion de version, imposant le verrouillage avant d'enregistrer des modifications, utilisent le modèle verrouillage-modification-déverrouillage. Ceux qui ne le font pas utilisent le modèle dit de copie-modification-fusion. Vous trouverez dans le chapitre 2 [1] du livre *Version control with subversion* une excellente explication détaillée et une comparaison de ces deux modèles. En général, le modèle copie-modification-fusion est plus adapté au développement Open Source, et tous les logiciels de gestion de versions abordés dans ce livre prennent en charge ce modèle.

Choisir un logiciel de gestion de versions

Au moment de l'écriture de ces lignes, le logiciel de gestion de versions libre le plus populaire est le Concurrent Versions System ou CVS [2]. Il y a longtemps maintenant qu'existe CVS. La plupart des développeurs expérimentés en sont déjà familiers, il fait plus ou moins ce que vous voulez, et puisque c'est le choix par défaut, il n'y aura guère de longs débats sur la validité de votre choix. CVS a cependant quelques inconvénients : il ne propose pas de se référer de manière simple à des modifications de plusieurs fichiers, il ne vous permet pas de renommer ou de copier des fichiers sous gestion de versions (et donc si vous avez besoin de réorganiser l'arbre de votre code après avoir lancé le projet, cela peut être très pénible), les possibilités de fusion laissent à désirer, il ne gère pas très bien les fichiers volumineux ou les fichiers binaires, et certaines opérations sont lentes quand elles concernent un grand nombre de fichiers.

Aucun de ces défauts n'étant rédhibitoire, il reste plutôt populaire. Cependant, depuis quelques années, Subversion, plus récent, marque des points particulièrement auprès des nouveaux projets [3]. Si vous démarrez, je vous conseille d'opter pour Subversion.

Mais d'un autre côté, étant impliqué dans le développement de Subversion, on peut raisonnablement mettre en doute mon objectivité. De plus, au cours des dernières années, de nouveaux logiciels de gestion de versions ont fait leur apparition. L'Annexe A dresse une liste de tous ceux que je connais. Comme cette liste le montre, le choix d'un logiciel de gestion de versions peut facilement devenir une quête sans fin. Il se peut que cette décision vous soit épargnée parce qu'elle sera prise par votre hébergeur. Mais si vous devez choisir, consulter vos développeurs, demandez-leur avec quel système ils ont l'habitude de travailler, puis choisissez-en un et utilisez-le. N'importe quel logiciel de gestion de versions stable et opérationnel fera l'affaire, votre choix n'est pas irréversible. Si vous n'arrivez pas à vous décider, choisissez alors CVS. Toujours le standard, il le restera certainement encore pour

1. http ://svnbook.red-bean.com/svnbook-1.0/ch02s02.html
2. http ://www.cvshome.org/
3. Voir http ://cia.vc/stats/vcs et http ://subversion.tigris.org/svn-dav-securityspace-survey.html, vous y trouverez des preuves de cette croissance.

des années. De plus, de nombreux systèmes gèrent la conversion depuis CVS, vous pourrez donc toujours changer d'avis ensuite.

Utilisation du logiciel de gestion de versions

Les recommandations, dans cette partie, ne se concentrent pas sur un logiciel de gestion de versions particulier : elles devraient être simples à mettre en œuvre dans n'importe lequel d'entre eux. Référez-vous à la documentation de votre système pour plus de détails.

Tout versionner

Non seulement vous devez garder le code source de votre projet sous gestion de versions, mais aussi : ses pages Web, la documentation, la FAQ et tout ce que les gens pourraient éditer. Conservez les proches du code source, dans le même arbre de dépôt. Ce qui est suffisamment important pour être écrit l'est aussi pour être « versionné », ce qui englobe donc toute information modifiable. Les objets invariables devraient être archivés, et non pas « versionnés ». Par exemple, un e-mail une fois envoyé ne change pas, donc le « versionner » est inutile (à moins qu'il fasse partie d'un plus grand document en constante évolution).

Il est important de tout « versionner » ensemble, au même endroit, afin que les gens n'aient à apprendre qu'un seul mécanisme pour envoyer des changements. Il arrive souvent qu'un contributeur débute en modifiant des pages Web ou de la documentation, puis plus tard, se met à apporter de petites contributions au code. Si le mécanisme est le même pour toutes les contributions, les participants ne doivent l'apprendre qu'une seule fois. Tout « versionner » ensemble permet aussi d'accompagner les nouvelles fonctionnalités d'une mise à jour de la documentation correspondante, et implique aussi qu'une création de branche du code créera également une branche dans la documentation, etc.

Ne gardez pas les fichiers générés sous gestion de versions. Ce ne sont pas vraiment des données éditables puisqu'elles ont été produites automatiquement à partir d'autres fichiers. Par exemple, certains systèmes de construction créent des configurations basées sur le modèle *configure.in*. Pour modifier la configuration, on devra alors éditer *configure.in* puis régénérer. En conséquence, seul le modèle *configure.in* est un « fichier éditable ». Ne « versionnez » que le modèle. Si vous « versionnez » également le fichier résultant, les gens oublieront inévitablement de le régénérer au moment de l'application d'une modification au modèle, et les incompatibilités causeront des confusions sans fin [1].

Il existe cependant une exception malheureuse à cette règle du « versionnage » de tout fichier éditable : le système de suivi de bogues. La base de données de bogues contient un grand nombre de données éditables, mais, pour des raisons techniques, ne peut pas ranger ces données dans le logiciel principal de gestion de versions (certains systèmes de suivi possèdent leur propre système de « versionnage » primitif, mais il est indépendant du dépôt principal du projet).

1. Alexey Makhotkin propose une vision différente du problème du versionnage des fichiers de configuration dans son message « configure.in et contrôle de version » consultable à l'adresse http ://versioncontrolblog.com/2007/01/08/configurein-and-version-control/.

Navigabilité

Le dépôt du projet devrait être navigable sur le Web. Ce qui veut dire que vous devez fournir à vos visiteurs, non seulement la possibilité de voir les dernières révisions des fichiers du projet, mais aussi de remonter le temps et visualiser les révisions précédentes, les différences entre les révisions, de lire les messages enregistrés pour certaines modifications, etc.

La navigabilité est importante : c'est le portail qui donne accès aux données du projet. Si le dépôt ne peut pas être vu grâce à un navigateur, et que quelqu'un veut accéder à un fichier particulier (pour voir si un certain correctif de bogue a été intégré par exemple), il devra d'abord installer le logiciel de gestion de versions localement, transformant de fait sa petite recherche de deux minutes en une tâche d'une demi-heure.

Des URL pour accéder à tout moment aux révisions spécifiques des fichiers et aux dernières révisions sont indispensables à une bonne navigabilité. Cela peut-être très utile au cours de discussions techniques ou lorsqu'on oriente les gens vers la documentation. Par exemple, au lieu de dire « Pour des conseils en vue de déboguer le serveur voir le fichier www/hacking.html dans votre copie de travail » on pourra dire « Pour des conseils en vue de déboguer le serveur, voir http ://svn.collab.net/repos/svn/trunk/www/hacking.html », donnant ainsi une URL qui renvoie toujours vers la dernière révision du fichier *hacking.html*. Il vaut mieux donner l'URL car, ainsi, vous levez toute ambiguïté, et la personne se réfère à la toute dernière version du document.

Certains systèmes de gestion de versions possèdent leurs propres mécanismes de navigation de dépôt intégrés, alors que d'autres s'appuient sur des programmes tiers. Trois de ces outils sont ViewCVS [1], CVSWeb [2] et WebSVN [3]. Le premier est compatible avec CVS et Subversion, le deuxième avec CVS seulement, et le troisième avec Subversion seulement.

E-mails de *commit*

Chaque ajout au dépôt devrait générer un e-mail précisant l'auteur des modifications, quels fichiers et dossiers ont été affectés, de quelle manière et quand. L'e-mail devrait être envoyé à une liste de diffusion spécialement faite pour les e-mails de *commit*, bien distincte des listes de diffusion où ce sont des personnes qui écrivent. Les développeurs et toutes les personnes intéressées devraient être encouragés à souscrire à cette liste car c'est le moyen le plus efficace de suivre ce qui se passe dans le projet au niveau du code. En plus des avantages techniques évidents en matière de vérification par les collègues (voir la section « Effectuez une inspection visible du code »), les e-mails de *commit* renforcent le sentiment de communauté parce qu'ils établissent un environnement partagé dans lequel les participants peuvent réagir à des évènements (les *commits*) qu'ils savent visibles de tous.

Selon votre logiciel de gestion de versions, la mise en place d'e-mails de *commit* sera différente, mais, en général, il existe un script ou d'autres outils sous forme de paquets pour le faire. Si vous avez du mal à le trouver, essayez de chercher de la documentation sur les

1. http ://viewcvs.sourceforge.net/
2. http ://www.freebsd.org/projects/cvsWeb.html
3. http ://Websvn.tigris.org/

mécanismes, en particulier sur les mécanismes post-*commit*, aussi appelés mécanismes loginfo dans CVS. Les mécanismes post-*commit* permettent d'automatiser certaines tâches en réponse à un *commit*. Le mécanisme, déclenché par un *commit*, engrange toutes les informations à propos de ce *commit* : il est ensuite libre d'utiliser ces informations pour réaliser de nombreuses tâches, comme par exemple envoyer un e-mail.

Avec un système d'e-mail de *commit* prêt à l'emploi, vous pouvez toujours modifier certaines options par défaut :

1. Certains systèmes d'envoi d'e-mail de *commit* n'incluent pas les *diffs* dans les e-mails, mais proposent, à la place, une URL pour voir les changements sur le Web en utilisant le système de navigation du dépôt. Même s'il est acceptable de donner une URL, on peut ainsi se référer à la dernière modification, et il est aussi très important que l'e-mail de *commit* comprenne les *diffs* complets. La lecture de l'e-mail fait déjà partie de la routine des participants, donc si le contenu de la modification est visible directement dans l'e-mail de *commit*, les développeurs peuvent l'examiner instantanément, sans sortir de leur messagerie. La plupart ne feront pas l'effort de cliquer sur un lien pour examiner quelque chose parce que cela implique une nouvelle action interrompant la continuité de leur activité en cours. De plus, si cette personne veut poser une question à propos de la modification, il est nettement plus simple d'appuyer sur le bouton « Répondre », et d'annoter simplement le *diff* cité, que de se rendre sur le site Web et copier/coller laborieusement des parties du *diff*, du navigateur Web vers la messagerie.

   (Évidemment, si le *diff* est très important, par exemple, au moment de l'ajout d'un gros morceau de nouveau code au dépôt, il paraît normal de ne pas inclure le *diff* et de mettre un lien. La plupart des systèmes d'envoi d'e-mail de *commit* peuvent appliquer ce genre de limitation automatiquement. Si le vôtre ne le permet pas, il vaut mieux inclure les *diffs* tout le temps, et accepter des mails énormes de temps à autre, plutôt que d'abandonner définitivement les *diffs*. Le développement coopératif repose sur l'aisance à modifier et commenter, c'est bien trop important pour s'en passer.)
2. Par défaut, l'option « Répondre à » pour les e-mails de *commit* devrait renvoyer à la liste de développement normale, pas à la liste d'e-mails de *commit*. Dans la pratique, quand quelqu'un examine un *commit* et écrit une réponse, celle-ci devrait être automatiquement envoyée à la liste de développeurs, là où les problèmes techniques sont normalement débattus. Voici quelques arguments en faveur ce choix. En premier lieu, il vaut mieux regrouper toutes les discussions techniques sur une même liste, parce que c'est là que les gens s'attendent à les voir, mais c'est aussi une manière de n'avoir qu'une seule archive à interroger en cas de recherche. En deuxième lieu, il se peut que des gens intéressés ne soient pas inscrits à la liste d'e-mails de *commit*. Ensuite, la liste de *commit* se présente comme un service de suivi des *commits*, pas pour suivre les *commits* et d'occasionnelles discussions techniques. Ceux qui se sont inscrits sur la liste d'e-mails de *commit* ne se sont pas abonnés à autre chose qu'un liste d'e-mails de *commit*. En leur envoyant d'autres informations par le biais de cette liste, vous dénaturez un contrat tacite. Enfin, les gens écrivent souvent des programmes qui lisent les listes d'e-mails de *commit*, et qui traitent les résultats (pour les afficher sur une page Web par exemple). Ces programmes sont fait pour

gérer des e-mails de *commit* automatiquement générés, pas des e-mails rédigés à la main.

Remarquez que ces conseils sur le « Répondre à » ne sont pas en contradiction avec les recommandations abordées précédemment. Il est préférable que l'expéditeur d'un message remplisse la case « Répondre à ». Ici, l'expéditeur est le logiciel de gestion de version lui-même, lequel configure le « Répondre à » afin d'indiquer que l'endroit approprié pour répondre est la liste de diffusion de développement, pas la liste de *commit*.

CIA : un autre mécanisme de publication des modifications
Les e-mails de commit ne sont pas l'unique relais des annonces de modifications. Depuis peu, un autre mécanisme appelé CIA (http ://cia.navi.cx/) est développé. CIA est un agrégateur et distributeur de statistiques en temps réel. Sa fonction principale est d'envoyer des notifications de *commit* aux canaux IRC afin que les personnes présentes soient informées du *commit* en temps réel. Bien que d'utilité technique moindre comparée aux e-mails de commit, les observateurs n'étant pas nécessairement présents quand l'information arrive sur IRC, cette technique a une utilité sociale énorme. Les gens ont le sentiment, grâce à ce système, de faire partie de quelque chose de vivant et d'actif, avec la possibilité d'en visualiser la progression en direct.
Cela fonctionne grâce au déclenchement du programme de notification de CIA par le mécanisme *post-commit*. Le programme de notification met les informations du *commit* sous forme d'un message XML, et l'envoie à un serveur central (généralement *cia.navi.cx*). Ce serveur distribue ensuite l'information de commit aux autres forums.
CIA peut également être configuré pour envoyer des flux RSS. Consulter la documentation à l'adresse http ://cia.navi.cx/ pour en savoir plus.
Pour voir une démonstration de CIA, rendez-vous sur *irc.freenode.net*, canal *#commit*s.

Utilisez les branches pour éviter les embouteillages

Les utilisateurs non-experts en gestion de versions ont parfois un peu peur de créer des branches et d'en fusionner. C'est certainement l'un des effets secondaire de la popularité de CVS : l'interface de CVS pour créer des branches, et les fusionner, est en quelque sorte contre-intuitive, beaucoup ont donc appris à éviter complètement ces opérations.

Si vous faites partie de ces personnes, il est urgent de vaincre cette peur et d'apprendre la création et la fusion de branches. Ces opérations ne sont pas difficiles une fois maitrisées, or elles prennent de plus en plus d'importance à mesure que de nouveaux développeurs se joignent au projet.

Les branches sont importantes parce qu'elles transforment une ressource rare, l'espace pour travailler dans le code du projet, en une ressource abondante. Normalement, tous les développeurs travaillent ensemble, dans le même bac à sable, à construire le même château de sable. Quand quelqu'un veut ajouter un nouveau pont-levis, mais qu'il n'arrive pas à convaincre tout le monde de l'amélioration potentielle, créer une branche lui permet de travailler dans son coin et de faire des essais. Si ses efforts sont couronnés de succès, il

peut inviter les autres développeurs à examiner son travail. Si tout le monde s'accorde à dire que le résultat est bon, il est possible de dire au logiciel de gestion de versions de déplacer (« fusionner ») le pont-levis du château personnel vers le château principal.

Cette liberté supplémentaire est évidemment essentielle dans le cadre du développement collaboratif. Les participants ont besoin d'espace pour essayer de nouvelles choses sans avoir peur d'interférer avec le travail des autres. De même, le code a parfois besoin d'être isolé de l'agitation du développement régulier pour que les développeurs puissent corriger un bogue ou stabiliser une version sans avoir l'impression de viser une cible en mouvement perpétuel (voir les sections « Stabiliser une version » et « Maintenir plusieurs lignes de sortie » dans le chapitre 7).

Utilisez les branches librement et encouragez les autres à en faire de même. Mais assurez-vous aussi qu'une branche donnée n'est active que pour le temps nécessaire, pas plus. Chaque branche active dilue l'attention de la communauté. Même ceux qui ne travaillent pas sur la branche gardent toujours un œil dessus pour savoir ce qu'il s'y passe. Une telle attention est bonne, bien sûr, et les e-mails de *commit* devraient être envoyés pour un *commit* dans une branche comme pour n'importe quel autre *commit*. Mais les branches ne devraient pas devenir un mécanisme qui disperse la communauté de développeurs. À de rares exceptions près, le destin d'une branche est d'être fusionnée à la ligne principale et de disparaître.

Unicité de l'information

De la possibilité de fusion découle un corollaire important : n'enregistrez jamais deux fois la même modification. Un changement donné ne devrait intégrer le logiciel de gestion de versions qu'une seule fois. À ce moment, la révision (ou l'ensemble de révisions), par laquelle la modification est entrée, est son unique identifiant. S'il doit être appliqué à d'autres branches (que celle par laquelle il est entré), il devrait être fusionné aux autres destinations à partir de son point d'entrée d'origine, plutôt que d'être enregistré comme un changement parfaitement identique. L'effet sur le code est le même, mais le suivi ou la gestion de cette modification deviendraient impossibles.

Concrètement, les implications sont différentes d'un logiciel de gestion de versions à l'autre. Dans certains systèmes, les fusions sont des évènements spéciaux, fondamentalement distincts des *commits*, et sont identifiés par leurs propres métadonnées. Dans d'autres, les résultats des fusions sont enregistrés de la même manière que les modifications, donc la première manière de distinguer un « *commit* fusionné » d'un « *commit* pour une nouvelle modification » est d'inspecter les messages enregistrés. Lors d'une fusion, le message enregistré ne doit pas répéter le message enregistré du changement d'origine. Indiquez simplement que c'est une fusion, et donnez l'identifiant de la révision du changement d'origine, avec, tout au plus, une phrase résumant les effets. Si quelqu'un désire voir le message enregistré en entier, il devrait consulter l'original.

Il est important d'éviter de répéter les messages enregistrés parce qu'ils peuvent être édités après avoir été enregistrés. Si un message enregistré est répété dans chaque cible de la fusion, et que le message d'origine est modifié les copies ne le sont pas, ce qui ne peut que mener à des incohérences.

Il en va de même pour l'annulation d'un changement. Si une modification est retirée du code, alors le message enregistré pour ce retour en arrière devrait simplement annoncer qu'une ou plusieurs révisions particulières ont été retirées, et non décrire le changement de code entraîné puisque ces informations peuvent être déduites de la lecture du message enregistré original. Évidemment, le message enregistré pour ce retour en arrière devrait également dire pourquoi ce changement a été retiré, mais il ne devrait rien copier du message enregistré originel. Si possible, éditez le message enregistré d'origine pour établir clairement qu'il a été retiré.

Toutes ces instructions impliquent que vous devriez utiliser une syntaxe cohérente lorsque vous vous référez à des révisions. Vous en verrez les bienfaits non seulement pour les messages enregistrés, mais aussi dans les e-mails, le système de suivi de bogues, etc. Si vous utilisez CVS, je vous conseille *path/to/file/in/project/tree :REV*, où REV est un numéro de révision dans CVS, comme par exemple « 1.76 ». Si vous utilisez Subversion, la syntaxe standard pour la révision 1729 est « r1729 » (le chemin d'accès n'est pas indispensable puisque Subversion utilise des numéros de révision généraux). Les autres systèmes emploient, en général, une syntaxe standard pour exprimer le nom de l'ensemble de modifications. Quelle que soit la syntaxe appropriée à votre système, encouragez les gens à l'utiliser lorsqu'ils parlent des modifications. La cohérence dans l'expression des noms des modifications permet un suivi du projet plus aisé (comme nous le verrons dans les chapitres 6 et 7). Étant donné qu'une grande partie de ce suivi sera faite par des volontaires, il doit être aussi simple que possible.

Autorisation

La plupart des logiciels de gestion de versions proposent une option permettant d'offrir aux participants le droit d'enregistrer des modifications dans des domaines précis du dépôt. En suivant le principe qu'avec un marteau en main, on se met à chercher des clous, nombreux sont les projets utilisant cette fonctionnalité sans réserve en accordant ces droits d'accès aux seuls domaines où il est autorisé d'enregistrer des modifications, tout en s'assurant qu'il est impossible de le faire nulle part ailleurs (voir la section « *Committers* » dans le chapitre 8 pour savoir comment les projets attribuent ces droits).

Si un contrôle strict ne gênera personne, des règles plus souples conviendront aussi. Certains projets fonctionnent avec un système d'honneur : quand on accorde l'accès de *commit* à une personne, même pour une sous-partie du dépôt, celle-ci reçoit en fait un mot de passe lui permettant d'enregistrer des modifications n'importe où dans le projet. Il lui est juste demandé de se limiter à ce domaine. Le risque n'est pas très grand, tous les *commits* sont examinés de toute façon. Si quelqu'un enregistre quelque chose qu'il n'était pas censé faire, d'autres le remarqueront et le signaleront. Tout est sous gestion de versions et si une modification doit être retirée il vous suffit de l'annuler.

L'approche plus souple possède quelques avantages. D'abord les développeurs s'ouvrent à d'autres domaines (ce qu'ils feront de toute façon s'ils s'impliquent durablement dans le projet), leur accorder des privilèges étendus ne demande pas plus de travail administratif. Une fois que la décision est prise, la personne peut commencer à enregistrer des modifications dans le nouveau domaine immédiatement.

Ensuite, l'ouverture peut être faite de manière plus progressive. En général, un *committer* dans le domaine X qui voudrait s'ouvrir au domaine Y commencera par envoyer des correctifs pour Y en demandant à ce qu'ils soient examinés. Si quelqu'un ayant déjà l'accès de *commit* dans le domaine Y voit ce correctif, et l'approuve, il peut simplement demander au requérant de l'enregistrer directement (en mentionnant évidemment le nom de celui qui approuve dans le message enregistré). Ainsi, le *commit* proviendra de la personne qui a effectivement fait la modification, ce qui est préférable à la fois du point de vue de la gestion de l'information et de celui de la reconnaissance.

Dernier point, et non des moindres, l'utilisation du système d'honneur encourage une atmosphère de confiance et de respect mutuel. Obtenir l'accès de *commit* à un sous-domaine est la reconnaissance de compétences techniques, cela signifie : « Nous remarquons que tu as les compétences pour faire des *commits* dans un certain domaine, nous te donnons le feu vert ». Mais en imposant un contrôle strict sur les autorisations vous dites : « Non seulement nous pensons que tes compétences ont des limites, mais nous avons aussi des doutes quant à tes intentions. » Ce n'est pas une déclaration que vous feriez si vous pouviez l'éviter. Donner l'accès de *commit* à quelqu'un, pour le faire entrer dans le projet, est une chance de lui ouvrir l'accès à un cercle de confiance mutuelle. Accorder plus de pouvoirs aux *committers* qu'ils ne sont censés avoir, et leur dire ensuite que c'est à eux de respecter leurs limites, voilà une bonne manière de procéder.

Le projet Subversion a adopté le système d'honneur depuis plus de quatre ans maintenant, avec 33 *committers* non-restreints et 43 *committers* aux droits restreints au moment où j'écris ces lignes. Le système ne distingue que ces deux catégories : ceux qui ont les droits de *committers* et ceux qui ne les ont pas, les sous-divisions plus précises ne sont établies que par les humains. Pourtant, jamais personne n'a dépassé les limites de son domaine. Une ou deux fois nous avons eu une légère incompréhension au niveau de l'étendue des privilèges de *commit*, mais ce problème a toujours été réglé rapidement et calmement.

Évidemment, dans les rares situations où l'autogestion est impraticable, vous devrez vous fier à un contrôle plus strict des autorisations. Cependant, devant des millions de lignes de code et des centaines ou des milliers de développeurs, un *commit*, pour n'importe quelle partie du code, devrait toujours être vérifié par ceux qui travaillent sur cette partie : ils pourront juger si quelqu'un a enregistré une modification alors qu'il n'aurait pas dû. Si une vérification régulière des *commits* n'est pas effectuée, le problème du système d'autorisation devrait être le cadet des soucis du projet.

Pour résumer, ne perdez pas trop de temps avec les autorisations du logiciel de gestion de versions, à moins que vous n'ayez des raisons particulières de le faire. Cela n'apporte en général pas d'amélioration tangible, et ne vous dispense pas des contrôles humains.

Il ne faut pas comprendre ici que les restrictions en elles-mêmes ne sont pas importantes bien sûr. Il serait dommageable pour le projet d'encourager les gens à enregistrer des modifications dans les parties où ils n'ont pas les compétences requises. De plus, dans de nombreux projets, un accès de commit non restreint possède un statut particulier : il va de pair avec le droit de vote sur les questions concernant le projet dans sa globalité. Cet aspect politique de l'accès de commit est discuté dans la section « Qui vote ? » du chapitre 4.

4. Suivi de bogues

Vaste sujet que le suivi de bogues : au travers de ce livre, nous en abordons différents aspects. Ici, je me concentrerai principalement sur l'installation et les considérations techniques, mais, avant d'aborder ces points, nous devons répondre à une question : quel genre d'informations doivent être exactement conservées dans le système de suivi de bogues ?

Le terme suivi de bogue est trompeur. Les systèmes de suivi de bogues sont aussi souvent utilisés pour les demandes de nouvelles fonctionnalités, des tâches ponctuelles, des correctifs non-sollicités : vraiment tout ce qui peut avoir un état final différent de l'état initial, avec des états de transitions optionnels, et qui accumule des informations au cours de son existence. Pour cette raison, les systèmes de suivi de bogues sont également appelés systèmes de suivi de problèmes, suivi de défauts, suivi des artéfacts, suivi de requêtes, file d'attente des problèmes, etc. Voir l'Annexe B, Systèmes de suivi de bogues libres pour une liste de logiciels.

Dans cet ouvrage, je continuerai à utiliser « système de suivi de bogues » pour désigner le logiciel qui s'occupe du suivi, parce que c'est ainsi que les gens l'appellent, mais je parlerai de problème pour désigner un objet particulier dans la base de données du système de suivi de bogues. Cela nous permet de faire la distinction entre le comportement, ou le mauvais comportement, auquel l'utilisateur a fait face (c'est à dire, le bogue en lui-même) et l'historique de découverte de bogues du système, du diagnostic et de la résolution finale. N'oubliez pas que, bien que la plupart des problèmes se rapportent à de vrais bogues, les problèmes peuvent être utilisés pour suivre d'autres types de tâches aussi.

Le cycle de vie classique d'un problème ressemble à ceci :

1. Quelqu'un enregistre le problème. Il fournit un résumé, une description basique (en indiquant comment reproduire le bogue si possible, voir la section nommée « Considérez tous les utilisateurs comme des volontaires potentiels » dans le chapitre 8 pour savoir comment encourager les rapports de bogue), et toute autre information demandée par le système de suivi. La personne enregistrant le problème peut être complètement étrangère aux rapports de bogues du projet, et les requêtes de fonctionnalités pourront aussi bien provenir de la communauté d'utilisateurs que des développeurs.

   Une fois enregistré, le problème est dans ce qu'on appelle un état ouvert. Parce qu'aucune action n'a encore été prise, certains systèmes le classent dans non vérifié et/ou non commencé. Il n'est assigné à personne, ou, dans certains systèmes, il est assigné à un utilisateur factice pour montrer qu'il n'est pas concrètement assigné. À partir de là, il est dans une zone d'attente : le problème a été enregistré, mais pas encore partie intégrante du projet.
2. D'autres lisent le problème, y ajoutent des commentaires et, si besoin est, demandent des clarifications sur certains points à celui qui a créé l'entrée.
3. Le bogue est reproduit. C'est peut-être le moment le plus important dans son cycle de vie. Même si le bogue n'est pas encore corrigé, le fait que quelqu'un d'autre que l'auteur de l'entrée ait été capable de le reproduire prouve son existence et confirme à l'auteur qu'il a contribué au projet en rapportant un vrai bogue, ce qui est tout aussi important.

4. Le bogue est diagnostiqué : ses causes sont identifiées et, si possible, l'effort requis pour le corriger est estimé. Assurez-vous que tout ceci est enregistré dans le problème, si la personne qui a fait le diagnostic du bogue doit soudainement quitter le projet pour un certain temps (comme cela arrive souvent avec les développeurs volontaires), quelqu'un d'autre pourra continuer le travail sur les bases de ce qui avait été fait.

   À cette étape, ou parfois dans les précédentes, un développeur peut « prendre possession » du problème et se l'assigner (la section intitulée « Distinguer clairement requête et assignation » dans le chapitre 8 éclaire plus en détails le processus d'assignation). La priorité du problème peut aussi être décidée à cette étape. Par exemple, un problème grave, au point de retarder la prochaine sortie, doit être identifié tôt, et le suivi devrait fournir un moyen de le noter.

5. Un calendrier est prévu pour la résolution du problème. Prévoir ne signifie par nécessairement donner une date à laquelle le problème sera corrigé. Parfois, c'est simplement qu'il a été décidé dans quelle future version (par forcément la prochaine) le bogue devrait être corrigé, ou qu'on a décidé qu'il n'était pas nécessaire de se fixer une version particulière. On peut se passer d'une prévision, si le bogue peut être corrigé rapidement.

6. Le bogue est corrigé (ou la tâche accomplie ou le correctif appliqué, peu importe). La modification, ou l'ensemble des modifications, qui l'a corrigé, devrait être enregistrée dans un commentaire du problème, après quoi le problème est clos et/ou marqué comme résolu.

Ce cycle connaît des variantes classiques. Il arrive qu'un incident soit clos très rapidement après avoir été enregistré, parce qu'il s'avère que ce n'est pas un bogue du tout, mais plutôt une mauvaise compréhension de la part de l'utilisateur. Ce genre de faux positif est de plus en plus fréquent à mesure qu'un projet fédère plus d'utilisateurs, et les développeurs les fermeront avec de moins en moins de tact. Essayez de contenir cette tendance. Cela n'apporte rien à personne, un utilisateur particulier n'est pas responsable des mauvais rapports précédents, seul les développeurs sont témoins de cette augmentation, pas les utilisateurs (dans la section nommée « Filtrer le système de suivi de bogues en amont » plus loin dans ce chapitre nous aborderons les techniques pour réduire le nombre de rapports non valides). Il faut voir également que, si plusieurs utilisateurs expriment sans arrêt la même incompréhension, il pourrait être bon de repenser cette partie du logiciel. Ce genre de répétition est détecté plus aisément si un responsable incident surveille la base de données de bogues, voir la section appelée « Responsable incidents » dans le chapitre 8.

Souvent un incident sera identifié comme étant un doublon, et sera clos peu après la première étape, c'est une autre variante fréquente du cycle de vie d'un incident. Un doublon est un incident enregistré par quelqu'un alors qu'il est déjà connu du projet. Les doublons ne sont pas confinés aux incidents ouverts : il est possible qu'un bogue revienne après avoir été corrigé (on appelle cela une régression), dans ce cas, la voie choisie est en général de ré-ouvrir l'incident d'origine, et de fermer tout nouveau rapport doublon du premier. Le système de suivi de bogue devrait garder en mémoire les relations dans les deux sens, afin que les informations de reproduction dans les duplicatas soient disponibles dans l'incident originel et vice versa.

Une troisième variation peut se produire quand les développeurs ferment l'incident, pensant avoir réglé le problème, dans le seul but que le rapporteur original rejette finalement la correction, et le présente à nouveau. C'est, en général, parce que les développeurs n'ont simplement pas accès aux ressources nécessaires pour reproduire le bogue, ou, parce qu'ils n'ont pas testé le correctif en employant la même recette de reproduction que le rapporteur. En plus de ces variations, d'autres petits détails du cycle de vie peuvent changer selon le logiciel de suivi. Mais de manière générale, c'est toujours pareil, et même si le cycle de vie n'est pas particulier aux logiciels Open Source, il a des implications sur l'usage que font les projets Open Source de leur système de suivi de bogues.

Comme l'étape 1 le laisse penser, le système de suivi est, au même titre que les listes de diffusion et les pages Web, la face visible du projet. N'importe qui peut rapporter un problème, n'importe qui peut jeter un œil à un incident, et n'importe qui peut parcourir la liste des incidents actuellement ouverts. Vous ne savez jamais, par conséquent, combien de personnes guettent les améliorations d'un problème particulier. Alors que la taille et les compétences de la communauté de développement déterminent la vitesse de résolution des problèmes, le projet devrait, au minimum, tenter de reconnaître les incidents à mesure qu'ils apparaissent. Même si le problème dure un moment, une réponse encourage le rapporteur à rester engagé car il voit son travail enregistré (souvenez-vous que rapporter un problème demande en général plus d'efforts que, par exemple, envoyer un e-mail). De plus, une fois un problème repéré par un développeur, le projet en prend conscience, dans le sens où le développeur peut guetter d'autres exemples, peut en parler aux autres développeurs, etc.

Les réponses rapides sont possibles à deux conditions :

– Le système de suivi doit être lié à une liste de diffusion afin que chaque changement apporté à un incident, y compris sa création, envoie un mail décrivant les modifications. Cette liste de diffusion est en général distincte des listes de développement habituelles puisque tous les développeurs ne veulent pas forcément recevoir des mails de bogues automatisés, mais (tout comme les mails de *commit*) le bandeau « Répondre à » devrait renvoyer vers la liste de développement.
– Le fichier d'envoi d'incident devrait inclure l'adresse e-mail du rapporteur afin de pouvoir lui demander plus d'informations (cependant, l'adresse e-mail ne devrait pas être obligatoire : certaines personnes préfèrent rapporter les bogues anonymement. Voyez dans ce chapitre la section nommée « Anonymat et engagement » pour plus de détails sur l'importance de l'anonymat).

Interaction avec les listes de diffusion

Faites en sorte que le système de suivi de bogues ne se transforme pas en un forum de discussion. S'il est important de garder une présence humaine dans ce système de suivi, il n'est pas exactement fait pour mener des discussions en temps réel. Essayez de le voir plutôt comme un archiveur, une façon d'organiser les faits et les références à d'autres discussions, principalement celles développées dans les listes de discussions.

Cette distinction est importante pour deux raisons. En premier lieu, le système de suivi de bogues est moins commode à utiliser que les listes de diffusion (ou que les salons de discussion en temps réel d'ailleurs). Ce n'est pas parce que les systèmes de suivi ont une interface

mal conçue, c'est simplement que leur interface a été conçue pour capturer et présenter des états discrets, pas des discussions continues. En second lieu, tous ceux devant être impliqués dans la discussion d'un problème particulier ne suivent pas forcément le système de suivi. Une bonne gestion des incidents demande que chaque problème soit porté à l'attention des bonnes personnes, il ne s'agit pas de demander à tous les développeurs de soutenir tous les problèmes (voir la section intitulée « Partager les tâches de gestion aussi bien que les tâches techniques » dans le chapitre 8). Au chapitre 6, dans la section « Pas de discussion dans le système de suivi de bogues », nous verrons comment s'assurer que les gens ne détournent pas accidentellement les discussions hors du bon forum vers ce système.

Certains systèmes de suivi de bogues peuvent surveiller les listes de diffusions, et enregistrer tous les e-mails se rapportant à un problème connu. Ils font en général cela en reconnaissant le numéro d'identification des incidents dans le sujet du mail, au sein d'un « string »* particulier. Les développeurs apprennent à inclure ces « strings » dans les e-mails pour attirer l'attention du système qui peut soit sauvegarder l'e-mail entier soit simplement enregistrer un lien vers l'e-mail dans l'archive habituelle de la liste de diffusion (cette solution est meilleure). Dans tous les cas, c'est une fonctionnalité très utile, si votre système la propose assurez-vous de l'activer et de rappeler aux utilisateurs de s'en servir.

Filtrer en amont le système de suivi de bogues

La plupart des bases de données d'incidents souffrent au final du même problème : un nombre écrasant de doublons, ou d'incidents invalides, enregistrés par des utilisateurs bien intentionnés mais inexpérimentés ou mal informés. La première chose à faire, pour combattre cette tendance, est en général, de mettre une note bien en vue sur la première page du système de suivi indiquant comment différencier un bogue de ce qui ne l'est pas, comment faire des recherches pour savoir s'il n'a pas déjà été enregistré et, finalement, comment en rapporter un de manière efficace si l'on pense toujours avoir affaire à un nouveau bogue.

Cela devrait vous donner un peu de répit, mais, avec l'augmentation du nombre d'utilisateurs, le problème reviendra. Individuellement, aucun utilisateur n'est responsable, il essaie simplement de contribuer au bien-être du projet, et même si un premier rapport de bogue n'est pas d'une grande aide, vous devez quand même encourager chacun à rester impliqué, et à mieux s'y prendre dans l'avenir. Parallèlement pourtant, le projet a besoin de garder une base de données d'incidents aussi propre que possible.

Deux choses vous aideront surtout à prévenir ce problème. D'abord, votre équipe, chargée de la surveillance du système de suivi de bogues, doit posséder les connaissances pour fermer les incidents invalides, ou les doublons, au fur et à mesure de leur arrivée. Ensuite, il faut obliger (ou encourager fortement) les utilisateurs à faire confirmer leurs bogues par une tierce personne avant l'enregistrement.

La première technique semble être largement répandue. Même les projets avec des bases de données énormes (par exemple, le système de suivi de bogues du projet Debian à l'adresse http ://bugs.debian.org/ qui contient 315 929 incidents au moment où j'écris) parviennent à faire vérifier chaque incident à son enregistrement. Plusieurs personnes peuvent assurer ce rôle selon le type d'incident. Par exemple, le projet Debian est un ensemble de paquets logiciels, donc Debian redirige automatiquement l'incident vers ceux qui sont en charge du

paquet. Évidemment, les utilisateurs peuvent parfois se tromper de catégorie quand ils enregistrent un bogue, l'incident sera donc envoyé en premier à la mauvaise personne, laquelle devra alors à son tour le rediriger. Malgré tout, l'important ici est que la charge de travail soit bien partagée (que l'utilisateur ait vu juste ou non au moment de l'enregistrement), la surveillance des incidents est toujours répartie plus ou moins équitablement entre les développeurs, permettant ainsi une réponse rapide.

La deuxième technique est moins répandue, sûrement parce qu'elle est plus difficile à automatiser. L'idée essentielle est que chaque nouvel incident doit être « parrainé » pour entrer dans la base de données. Quand un utilisateur pense avoir découvert un problème, on lui demande de le décrire dans une des listes de diffusion, ou dans un canal IRC, afin d'obtenir la confirmation de bogue de la part d'un tiers. Faire entrer ce second avis rapidement, peut vous éviter beaucoup de faux rapports. Parfois, la deuxième personne est capable de dire si ce comportement est un bogue ou non, ou s'il a été corrigé dans une version plus récente. Il peut aussi bien connaître les symptômes d'un bogue déjà identifié, et éviter un doublon en dirigeant l'utilisateur vers l'incident précédent. Il suffit souvent de simplement demander à l'utilisateur « As-tu fait une recherche dans le système de suivi de bogues pour voir s'il n'a pas déjà été rapporté ? ». Nombreux sont ceux qui n'y pensent pas, et qui feront de bon cœur cette recherche.

Le système de parrainage peut être très efficace pour maintenir en ordre la base de données, mais il a ses inconvénients également. De nombreuses personnes continueront de faire des rapports dans leur coin, au mépris des instructions permettant de trouver un parrain pour les nouveaux incidents, ou, parce qu'ils ne les auront pas vues. De plus, la plupart des nouveaux rapporteurs n'imaginent pas les efforts nécessaires pour maintenir une base de données d'incidents propre, il est donc injuste de les réprimander trop durement pour avoir ignoré les directives. Voilà pourquoi les volontaires doivent à la fois être vigilants et se montrer souples quand ils renvoient un incident non parrainé à son rapporteur. Le but est d'entraîner chaque rapporteur à utiliser le système de parrainage dans le futur, afin que le nombre de personnes comprenant le système de filtres des incidents augmente. Quand vous découvrez un incident non parrainé voici les étapes d'une réaction idéale :

1. Répondez immédiatement au problème, remerciez poliment l'utilisateur pour sa contribution, mais dirigez le vers les règles de parrainage (qui devraient évidemment être bien en évidence sur le site Web).
2. Si l'incident est valide et n'est pas un doublon, approuvez-le malgré tout, et lancez-le sur son cycle de vie normal. Après tout la personne ayant fait le rapport est maintenant au courant du parrainage, il n'y a donc pas de raison de perdre le travail accompli en fermant un rapport valide.
3. Si la validité du rapport n'est pas évidente, fermez-le, mais demandez au rapporteur de le rouvrir s'il obtient la confirmation d'un parrain. Quand il l'obtient, il devrait mettre une référence au sujet pour lequel il a obtenu confirmation (c'est à dire, une URL des archives de la liste de diffusion).

Ne pas oublier que ce système améliorera, avec le temps, le rapport signal/bruit dans la base de données des incidents, mais qu'il ne fera pas complètement disparaître les mauvais rapports. Ce n'est qu'en interdisant à tous, excepté les développeurs, l'accès au système de

suivi de bogues que vous les ferez disparaître intégralement : un remède bien souvent pire que le mal. Mieux vaut accepter que la résolution de problèmes tangents fera toujours partie de l'entretien routinier du projet, et essayer d'obtenir autant d'aide extérieure que possible. Voyez aussi la section appelée « Responsable incident » dans le chapitre 8.

5. IRC et les chats en temps réel

De nombreux projets proposent des salons de discussion en temps réel par le biais d'IRC (Internet Relay Chat), ou encore de forums où utilisateurs et développeurs peuvent poser des questions et recevoir des réponses rapidement. N'hébergez pas de serveur IRC sur votre propre site Web, même si vous en avez la possibilité, ça n'en vaut pas la peine en général. Faites plutôt comme tout le monde : hébergez votre canal IRC chez Freenode [1]. Freenode vous donne le contrôle nécessaire pour administrer les canaux IRC de votre projet [2], tout en vous déchargeant de l'entretien assez conséquent d'un serveur IRC.

La première chose à faire est de choisir un nom de canal. Le choix le plus évident est le nom de votre projet, s'il est disponible chez Freenode, utilisez-le. Si ce n'est pas le cas, essayez de choisir quelque chose de facile à retenir, aussi proche que possible du nom du projet. Mettez en avant l'existence du canal sur le site Web de votre projet, ainsi un visiteur ayant une question le verra tout de suite. Voici, par exemple, le message dans un cadre bien en vue en haut de la page de Subversion :

> *« Si vous utilisez Subversion, nous vous recommandons de rejoindre la liste de diffusion* users@subversion.tigris.org *et de lire Subversion Book [http ://svnbook.red-bean.com/] ainsi que la FAQ [http ://subversion.tigris.org/faq.html]. Vous pouvez également poser vos questions sur IRC à irc.freenode.net canal #svn. »*

Certains projets utilisent plusieurs canaux, un par sous-sujet. Ce peut être, par exemple, un canal pour les problèmes d'installation, un autre pour les questions relatives à l'utilisation, un autre pour discuter du développement, etc. (la partie appelée « Gérer la croissance » dans le chapitre 6 aborde cette division en plusieurs canaux). Tant que votre projet est jeune, il ne devrait y avoir qu'un seul canal où tout le monde se parle. Plus tard, quand le rapport utilisateur/développeur augmente, créer des canaux séparés pourra devenir nécessaire.

Comment connaître tous les canaux disponibles et savoir où poser les questions ? Comment les utilisateurs connaîtront-ils les règles en vigueur pour s'exprimer ?

Il faut le leur dire en créant un message d'accueil [3]. Il s'agit d'un court texte qui indique brièvement les règles aux nouveaux venus ; il est vu par tous les utilisateurs entrant dans un canal et fournit des liens pour en apprendre davantage. Par exemple :

1. http ://freenode.net/
2. Vous n'êtes pas obligé de faire un don à Freenode, mais, si votre projet peut se le permettre, je vous invite à y réfléchir : il est déductible de l'impôt sur le revenu.
3. Pour configurer un message d'accueil utilisez la commande /topic. Toutes les commandes dans IRC commencent par « / ». Si vous souhaitez en apprendre plus sur l'utilisation d'IRC et son administration, lisez en particulier le très bon tutoriel sur la page http ://www.irchelp.org/. http ://www.irchelp.org/irchelp/irctutorial.html

Vous discutez maintenant dans #svn

Le sujet de #svn est Forum concernant les questions des utilisateurs de Subversion, voir également http ://subversion.tigris.org/. Les discussions relatives au développement se font dans #svn-dev. S'il vous plait, n'y collez pas vos transcriptions, utilisez plutôt un site comme pastebin : http ://pastebin.ca.
NOUVEAUTÉS : Subversion 1.1.0 est terminé,
voir http ://svn110.notlong.com pour en savoir plus.

C'est succinct, mais un nouvel arrivant y apprend tout ce qu'il doit savoir. On connait le rôle exact du canal, l'adresse du site du projet (au cas où quelqu'un s'égare sur le canal, sans être d'abord passé par le site du projet), les autres canaux sont mentionnés et l'utilisateur y découvre les consignes à propos du collage (*pasting*).

Sites de collage
Un canal IRC est un espace commun : tout le monde peut voir ce que chacun dit. Normalement, c'est une bonne chose, puisque cela permet aux gens de s'engager dans une conversation s'ils pensent pouvoir y apporter quelque chose, mais aussi d'apprendre en tant que spectateurs. Par contre, cela devient problématique lorsqu'un participant doit fournir une grande quantité d'information en un bloc, comme par exemple la transcription d'une session de débogage. En effet, coller trop de lignes dans le canal gênera les autres conversations.
L'alternative est d'utiliser un site de *pastebin* ou *pastebot* [NdT : poubelle à collage ou robot à collage littéralement].
Quand vous avez besoin que l'on vous fournisse une quantité importante de texte, demandez-leur, non pas de le coller dans le canal, mais plutôt de se rendre (par exemple) sur http ://pastebin.ca/, d'y coller leurs données et de donner l'URL résultante dans le canal IRC. Chacun peut alors visiter le lien, et avoir accès aux données.
Il existe maintenant beaucoup de sites de collage, bien trop pour en dresser une liste exhaustive, mais en voici quelques-uns parmi les plus utilisés :

- http ://www.nomorepasting.com
- http ://pastebin.ca
- http ://nopaste.php.cd
- http ://rafb.net/paste/
- http ://sourcepost.sytes.net
- http ://extraball.sunsite.dk/notepad.php
- http ://www.pastebin.com

Robots

Beaucoup de canaux IRC orientés technique possèdent un membre non-humain, appelé robot, capable d'enregistrer et de régurgiter des informations en réponse à des commandes particulières. On s'adresse au robot comme à n'importe quel autre membre du canal, c'est-à-dire que les commandes sont déclenchées en « lui parlant ». Par exemple :

```
  <kfogel> ayita: apprend diff-cmd =
http://subversion.tigris.org/faq.html#diff-cmd
<ayita> Merci!
```

Cela indique au robot (qui participe au canal sous le nom de ayita) de se souvenir d'une certaine URL en réponse à la requête « diff-cmd ». Maintenant on peut commander ayita et demander au robot de donner des informations à un autre utilisateur à propos de la commande diff-cmd :

```
  <kfogel> ayita: parle à a.nonyme de diff-cmd
<ayita> a.nonyme: http://subversion.tigris.org/faq.html#diff-cmd
```

La même chose peut-être accomplie via un raccourci pratique :

<kfogel> !a a.nonyme diff-cmd
<ayita> a.nonyme : http ://subversion.tigris.org/faq.html#diff-cmd

L'ensemble des commandes exactes et des comportements diffèrent d'un robot à l'autre. L'exemple donné ci-dessus est celui d'ayita [1], lequel tourne généralement sur *#svn* chez freenode. On peut également citer Dancer [2] et Supybot [3]. Sachez qu'aucun privilège sur le serveur n'est nécessaire pour utiliser un robot. Un robot est un programme client, n'importe qui peut en installer un, et le configurer pour qu'il suive un serveur ou un canal particulier.

Si vous retrouvez sans arrêt les mêmes questions sur votre canal, je vous recommande vivement d'installer un robot. Seul un faible pourcentage des utilisateurs du canal acquerront les compétences requises pour manipuler le robot, mais ils pourront répondre à bien plus de questions grâce à son efficacité.

Archiver les conversations

Bien qu'il soit possible d'archiver tout ce qu'il se passe dans un canal IRC, on n'attend pas de vous, en général, que vous le fassiez. Les conversations IRC ont beau être publiques, beaucoup de gens les voient comme des conversations informelles semi-privées. Les utilisateurs ne prêteront pas forcément attention à la grammaire, et feront souvent part de leur opinion (à propos par exemple d'autres logiciels ou d'autres programmeurs), ils ne veulent pas forcément que tout cela soit conservé *ad vitam æternam* dans une archive en ligne.

Bien sûr, il y aura des cas de figure où des citations devraient être préservées, ce qui est normal. La plupart des clients IRC peuvent enregistrer une conversation dans un fichier à la demande de l'utilisateur, et, si ce n'est pas le cas, on peut simplement copier/coller une conversation IRC sur un forum permanent (le plus souvent dans le système de suivi de bogues). Mais un enregistrement hasardeux peut mettre certains utilisateurs mal à l'aise. Si vous archivez tout, assurez-vous de l'annoncer clairement dans l'introduction du canal, et donnez le lien vers les archives.

1. http ://hix.nu/svn-public/alexis/trunk/
2. http ://dancer.sourceforge.net/
3. http ://supybot.com/

6. Les flux RSS

RSS (Really Simple Syndication) est un mécanisme permettant de distribuer des résumés de nouvelles aux « abonnés », c'est à dire aux personnes qui ont manifesté le désir de recevoir ces résumés. Une source RSS donnée est, généralement, appelée un flux, et les abonnés y ont accès par ce qu'on appelle un lecteur de flux ou un agrégateur de flux. RSS Bandit [1] ou le logiciel éponyme FeedReader [2] sont deux exemples de lecteurs de flux Open Source.

Nous n'avons pas ici la place pour une explication technique détaillée du fonctionnement des flux RSS [3], mais il y a deux choses que vous devriez savoir. En premier lieu, c'est l'abonné qui choisit son lecteur de flux, et c'est le même pour tous les flux suivis par l'abonné. En fait, c'est un des principaux arguments des lecteurs de flux RSS : l'abonné choisit une interface pour lire tous ses flux, chaque flux pouvant se concentrer uniquement sur la distribution de contenu. En deuxième lieu, les flux RSS sont maintenant omniprésents, à tel point que la majorité des utilisateurs ne savent même pas qu'ils les utilisent. Pour monsieur Toutlemonde, RSS est un petit bouton sur une page Web avec une petite ligne annonçant « s'abonner à ce site » ou « Flux de nouvelles ». Il suffit d'appuyer sur ce bouton pour que votre lecteur de flux (parfois intégré à votre page d'accueil) se mette automatiquement à jour quand le site publie une nouvelle.

Tout ça pour dire que les projets Open Source devraient proposer un flux RSS (les forges en général le font automatiquement, voir la section « Forges »). Prenez toutefois garde à ne pas publier trop de nouvelles quotidiennement afin que vos abonnés puissent séparer le bon grain de l'ivraie. Si les mises à jour sont trop fréquentes, les gens finiront simplement par ignorer le flux ou se désabonner. L'idéal pour un projet est de proposer des flux distincts. Un pour les annonces importantes, un autre qui suivrait, par exemple, les évènements du suivi de bogues, un autre pour chaque liste de diffusion, etc. En pratique, ça n'est pas si simple à mettre en place correctement : les visiteurs du site du projet comme les administrateurs pourraient être un peu perdus. Mais le projet devrait fournir *a minima* un flux RSS pour la page d'accueil du site, afin d'envoyer les annonces importantes telles que sorties et alertes de sécurité [4].

7. Les wikis

Un wiki est un site Web permettant à quiconque d'en éditer ou d'en enrichir le contenu, le terme "wiki" (qui provient de l'hawaïen "rapide" ou "super vite") sert aussi à désigner les logiciels autorisant de telles modifications. Les wikis ont été inventés en 1995, mais leur popularité n'a vraiment décollé que depuis 2000 ou 2001, entraînée en partie par le succès de

1. http ://www.rssbandit.org/
2. http ://www.feedreader.com/
3. Voir http ://www.xml.com/pub/a/2002/12/18/dive-into-xml.html à ce propos.
4. Il faut rendre à César ce qui appartient à César : cette section n'apparaissait pas dans la version originale du livre, mais l'article sur le blog de Brian Aker « Release Criteria, Open Source, Thoughts On… » m'a rappelé l'importance des flux RSS pour les projets Open Source.

Wikipédia [1], une encyclopédie libre basée sur un wiki. Le wiki se situe à mi-chemin entre une page Web et un canal IRC : les wikis n'évoluent pas en temps réel, les utilisateurs peuvent donc peser leurs mots et travailler leurs contributions, mais étant également vraiment très facile à éditer, on s'affranchit de toute la partie interface liée à une page Web classique.

Si vous décidez d'utiliser un wiki, concentrez vos efforts sur la mise en page, afin qu'elle soit claire et visuellement attirante pour que les visiteurs (c'est-à-dire les éditeurs potentiels) sachent instinctivement comment y intégrer leurs contributions. De même, postez ces standards sur le wiki lui-même, afin que les gens aient un guide auquel se référer. Ne croyez pas naïvement comme tant d'administrateurs de wiki que la somme de contributions individuelles de qualité sera automatiquement de qualité. Les sites Web ne fonctionnent pas ainsi. Chaque écrit individuel, ou paragraphe, considéré séparément peut être bon, mais ne le sera pas noyé dans un tout désorganisé et confus. Trop souvent les wikis souffrent des défauts suivants :

Manque de principe de navigation — Grâce à un site bien organisé, les visiteurs ne se sentent jamais perdus. Si les pages sont bien conçues, par exemple, les lecteurs feront intuitivement la différence entre une zone « sommaire » et une zone « contenu ». Ceux qui participent au wiki respecteront ces différences également, mais seulement si elles sont présentes dès le départ.

Doublons — Des pages au contenu semblable apparaîtront immanquablement parce que les participants isolés n'avaient pas forcément remarqué le doublon. Ceci peut être, en partie, dû au manque de principes de navigation cités précédemment. Les contributeurs peuvent, en effet, ne pas trouver le doublon s'il ne se trouve pas là où ils l'attendent.

Incohérence du public visé — Ce problème est inévitable quand il y a trop d'auteurs, mais il peut être atténué si vous proposez des indications pour la création de contenu nouveau. Il peut aussi être utile d'éditer fermement les nouvelles contributions au départ, pour montrer l'exemple, afin que les normes commencent à s'imposer.

La solution commune à tous ces problèmes est la même : vous devez avoir des règles éditoriales et en faire la démonstration, non seulement en les affichant, mais aussi en les appliquant lors de l'édition de pages. En général, les wikis amplifient les défauts des exemples de base puisque les participants imitent les modèles donnés. Vous ne pouvez pas vous contenter de mettre en place le wiki en espérant que tout s'arrange tout seul. Vous devez aussi l'amorcer avec du contenu bien rédigé afin que les rédacteurs aient un modèle à suivre.

Un très bon exemple de wiki bien géré est Wikipédia. Il faut dire aussi que le contenu (les entrées encyclopédiques) se prête naturellement au format wiki. Mais si vous examinez Wikipédia de plus près, vous remarquerez que ses administrateurs ont posé des bases très solides pour favoriser la coopération. Vous y trouverez une documentation bien détaillée sur l'écriture de nouvelles entrées, la manière de conserver un point de vue neutre, quel genre d'édition effectuer, lesquelles sont à éviter, un processus de résolution des conflits sur les éditions (incluant plusieurs étapes, jusqu'à l'arbitrage final), etc. Les administrateurs possèdent aussi un contrôle d'autorisation ; ainsi, si une page est souvent vandalisée, ils peuvent en interdire l'édition jusqu'à la résolution du problème. En d'autres termes, ils ne se

1. http ://www.wikipedia.org/

sont pas contentés de balancer des modèles sur un site Web en rêvant. Wikipédia fonctionne car ses fondateurs ont bien réfléchi à la manière d'amener des milliers d'inconnus à modifier leurs contributions selon une vision commune. Sans nécessairement atteindre ce niveau de préparation pour gérer un wiki de projet de logiciel libre, il est bon de s'en inspirer.

Pour de plus amples informations à propos des wikis, voir la page Wikipedia qui leur est consacrée [1]. Précisons que le premier wiki existe toujours, reste en bonne santé et contient de nombreuses discussions sur la manière de s'occuper d'un wiki [2].

8. Les sites Web

Il y a peu de chose à dire, d'un point de vue technique, à propos de la création du site Web du projet : rédaction et mise en page sont des tâches relativement simples, dont la plupart des points importants ont été abordés dans le chapitre précédent. Sa principale fonction est de présenter le projet de manière claire et accueillante, et de faire le lien entre les outils (le logiciel de gestion de versions, le système de suivi de bogues, etc.). Si vous n'avez pas les connaissances nécessaires à la mise en place d'un serveur Web, il n'est, en général, pas compliqué de trouver quelqu'un sachant le faire qui vous aidera avec plaisir. Néanmoins, pour économiser du temps et des efforts, on préfère souvent utiliser des sites d'hébergement ou forges.

Les forges

L'utilisation de forges possède deux gros avantages. Le premier est la capacité du serveur et la bande passante, ces serveurs sont des grosses boîtes reliées à des tuyaux vraiment bien larges. Peu importe le succès rencontré par votre projet, vous ne risquez pas d'avoir de problème d'espace disque ou de dépasser les limites de votre connexion. Le deuxième avantage est la simplicité : les serveurs sont livrés avec un système de suivi de bogues, un logiciel de gestion de versions, un gestionnaire de liste de diffusion, un archiveur et tout ce dont vous avez besoin pour gérer un site. Les outils sont pré-configurés et la sauvegarde des données stockées dans les outils est gérée. Vous n'avez que peu de décisions à prendre. On vous demande simplement de remplir un formulaire, d'appuyer sur un bouton et voilà, vous avez le site de votre projet.

Les forges présentent des avantages plutôt appréciables. Leur inconvénient : vous devez accepter leurs choix et leurs configurations, même si une autre solution serait mieux adaptée à votre projet. En général, les sites tout compris peuvent être quelque peu paramétrés, mais vous n'aurez jamais un contrôle aussi précis que si vous mettiez en place le site vous-même en ayant un accès complet à l'administration du serveur.

Un très bon exemple de ceci est la gestion des fichiers générés. Certaines pages de projets peuvent être des fichiers générés, il existe des systèmes pour garder les FAQ dans un format maître facilement éditable, à partir duquel des fichiers HTML, PDF (ou encore d'autres

1. http ://fr.wikipedia.org/wiki/Wiki
2. Voir http ://www.c2.com/cgi/wiki ?WelcomeVisitors (/wiki ?Why WikiWorks et /wiki ?WhyWikiDoesntWork), vous y trouverez plusieurs points de vue.

formats) peuvent être générés. Comme déjà détaillé précédemment dans la section appelée « Tout versionner », il n'est pas souhaitable de numéroter la version des formats générés, seulement celle du fichier maître. Mais quand votre site Web est hébergé sur le serveur d'une autre personne, il peut se révéler impossible de mettre en place un système régénérant la version HTML en ligne de la FAQ dès que le fichier maître est modifié. La seule solution est de versionner également les formats générés afin qu'ils apparaissent sur le site Web.

Les conséquences peuvent être des plus importantes. Vous pourriez ne pas avoir autant de contrôle que souhaité sur la présentation. Certaines de ces forges vous permettent de personnaliser vos pages Web, mais la configuration par défaut du site finit habituellement par apparaître maladroite. Par exemple, certains projets qui s'hébergent eux-mêmes sur SourceForge ont des pages complètement personnalisées, mais placent toujours des liens vers leur « page chez SourceForge » pour plus d'informations. La page chez SourceForce représente ce qu'aurait été la page du projet si le projet n'avait pas utilisé une page personnalisée. La page chez SourceForge contient des liens vers le système de suivi de bogues, le dépôt CVS, les téléchargements, etc. Malheureusement, une page chez SourceForge contient également une bonne quantité de contenu étranger. En haut de la page se trouve un bandeau publicitaire, souvent une image animée. À gauche, on voit listés verticalement des liens sans rapport pour ceux intéressés par le projet. Le côté droit est affublé d'une autre publicité. Seul le centre de la page est dédié réellement au projet, mais même cette partie est ordonnancée de manière confuse ; en conséquence, les visiteurs ne sont jamais sûrs de l'endroit où cliquer.

Chaque petit détail du design de SourceForge est bien pensé, en tout cas de leur point de vue, il en est ainsi des emplacements publicitaires. Mais pour notre projet, le résultat peut être une page loin de l'idéal. Je ne m'en prends pas spécifiquement à SourceForge, les mêmes remarques s'appliquent à de nombreuses forges. Je veux simplement dire que tout n'est pas rose. Vous vous déchargez des tâches techniques de la gestion du site du projet, mais en contrepartie vous sacrifiez votre contrôle total.

Vous êtes le seul à pouvoir décider si les forges représentent le meilleur choix pour votre projet. Si vous choisissez une forge, gardez-vous la possibilité de migrer sur vos propres serveurs plus tard, en utilisant, comme adresse du projet, un nom de domaine personnel. Vous pouvez rediriger l'URL vers le site hébergé ou avoir votre page complètement personnalisée sur cette URL publique, et rediriger les utilisateurs vers la forge pour les fonctionnalités plus sophistiquées. Assurez-vous simplement de faire en sorte que, si vous optez pour une autre solution d'hébergement par la suite, l'adresse du projet n'ait pas besoin de changer.

Choisir un hébergeur

Le plus important et le mieux connu des sites d'hébergement est SourceForge [1]. Deux autres sites proposent des services identiques ou similaires : savannah.gnu.org [2] et Ber-

1. http ://www.sourceforge.net/
2. http ://savannah.gnu.org/

liOS.de [1]. Quelques organisations, comme l'Apache Software Foundation [2] et Tigris.org [3] [4], offrent un hébergement aux projets qui s'inscrivent dans leur mission et leur communauté de projets existants.

Haggen So a effectué une évaluation minutieuse de différentes forges, au cours de ses recherches pour sa thèse de doctorat, *Construction of an Evaluation Model for Free/Open Source Project Hosting (FOSPhost) sites* [5]. Les résultats se trouvent sur ibiblio.org [6], je vous renvoie particulièrement à son graphique comparatif [7].

Anonymat et engagement

Un problème qui n'est pas spécifique aux forges, mais que l'on y retrouve souvent, est le détournement des fonctionnalités d'identification. La fonctionnalité en elle-même est très simple : le site permet à chaque visiteur de s'enregistrer grâce à un nom d'utilisateur et un mot de passe. À partir de ce moment, il conserve un profil pour cet utilisateur, et les administrateurs du projet peuvent lui attribuer certaines permissions, par exemple, le droit de faire des ajouts au dépôt.

Cela peut-être très utile et c'est d'ailleurs l'un des principaux avantages des forges. Le problème étant que l'identifiant de l'utilisateur est requis pour des tâches qui devraient être permises aux visiteurs non enregistrés, en particulier pour ajouter un rapport dans le système de suivi de bogues ou pour commenter des problèmes existants. En demandant un nom d'utilisateur à l'enregistrement en vue de telles actions, le projet demande plus d'engagement pour des tâches se devant d'être rapides et aisées. Évidemment, on voudrait pouvoir contacter une personne ayant entré des données dans le système de suivi de bogues, mais il suffit d'avoir un champ où entrer son adresse e-mail (s'il le désire). Si un nouvel utilisateur détecte un bogue et veut le rapporter, devoir remplir un formulaire pour se créer un compte avant de pouvoir entrer le bogue dans le système de suivi ne peut que le décourager. Il pourrait simplement décider, finalement, de ne pas rapporter le bogue.

Les avantages de la gestion des utilisateurs compensent généralement ces inconvénients. Cependant, s'il est possible de choisir les actions pouvant être effectuées anonymement, assurez-vous que, non seulement les visiteurs non-enregistrés ont l'accès aux ressources en lecture seule, mais aussi la possibilité d'entrer des données, en particulier dans le système de suivi de bogues et, si vous en avez un, dans le wiki.

1. http ://www.berlios.de/
2. http ://www.apache.org/
3. http ://www.tigris.org/
4. Avertissement : je suis employé par CollabNet qui finance Tigris.org, et j'utilise Tigris régulièrement.
5. NdT : *Construction d'un modèle d'évaluation des sites d'hébergement de projets libres/Open Source*.
6. http ://www.ibiblio.org/fosphost/
7. http ://www.ibiblio.org/fosphost/exhost.htm

Infrastructure sociale et politique

Certaines interrogations concernant les logiciels libres sont récurrentes : « Comment ça marche ? Qu'est-ce qui fait avancer un projet ? Qui prend les décisions ? ». Les réponses invoquant la méritocratie ne me satisfont jamais, pas plus que l'esprit de coopération, le code qui parle pour lui-même, etc. Le fait est que la réponse à ces questions n'est pas simple. La méritocratie, la coopération et le code qui marche entrent bien sûr en ligne de compte, mais n'expliquent pas vraiment comment un projet fonctionne au quotidien et encore moins comment les conflits sont résolus.

Ce chapitre tente de mettre en avant les fondements structurels communs aux projets réussis. J'emploie « réussis » non seulement pour parler de qualité technique, mais aussi de santé opérationnelle et de capacité de survie. Un projet en bonne santé sait incorporer en continu de nouvelles contributions au code, de nouveaux développeurs, et être réactif face aux rapports de bogue. Quant à la capacité de survie, c'est l'aptitude du projet à exister indépendamment de tout participant individuel ou sponsor, autrement dit à continuer d'être même si tous les membres fondateurs passent à autre chose. La réussite technique n'est pas difficile à atteindre mais, sans une équipe de développeurs et des fondations sociales solides, un projet peut se montrer incapable de faire face à la croissance que génèrent des débuts réussis ou aux départs d'individus charismatiques.

Les façons d'atteindre cette réussite sont diverses. Certaines impliquent une structure de gouvernance formelle qui permet de résoudre les débats, d'intégrer (et parfois d'exclure) de nouveaux développeurs, de planifier de nouvelles fonctionnalités, etc. D'autres impliquent une structure moins formelle, mais plus d'auto-modération volontaire. Le mode de gouvernance repose alors sur le respect de règles fiables. Ces deux modes de gestion mènent au même résultat : un sens de la continuité institutionnelle, étayé par des habitudes et des procédures bien comprises de tous ceux qui participent au projet. Ces caractéristiques sont même

plus importantes encore dans des systèmes auto-organisés que dans ceux où le contrôle est centralisé, car dans l'auto-organisation, chacun est conscient que quelques pommes pourries peuvent gâter tout le panier, du moins pendant un certain temps.

1. Fourchabilité

L'ingrédient indispensable qui maintient les développeurs unis autour d'un projet de logiciel libre et les amène à faire, si nécessaire, des compromis est la « fourchabilité » du code, c'est-à-dire la possibilité offerte à chacun de prendre une copie du code source et de l'utiliser pour démarrer un projet concurrent, connu sous le nom de « fork » ou « fourche » en français. Paradoxalement la possibilité de créer une fourche est généralement un levier beaucoup plus puissant dans les projets de logiciel libre que les vraies fourches, qui restent très rares. Une fourche n'est bonne pour personne (pour des raisons exposées plus en détail dans la section « Fourches » du chapitre 8). Plus la menace de bifurcation devient sérieuse, plus chacun est prêt à faire des compromis pour l'éviter.

La fourche, ou plutôt la possibilité de démarrer une fourche, est la raison pour laquelle il n'existe pas de vrai dictateur dans les projets de logiciels libres. Cette déclaration peut paraître surprenante, étant donné qu'il est très courant d'entendre parler d'un « dictateur » ou d'un « tyran » au sein d'un projet Open Source. Mais il s'agit d'une sorte de tyrannie particulière, très loin de la définition classique de ce terme. Imaginez un roi dont les sujets pourraient à tout moment copier le royaume entier et emménager dans la copie du royaume pour le gouverner comme bon leur semble. Il est quasi certain que le règne d'un tel roi serait fort différent de celui dont les sujets sont contraints de rester sous sa loi quoi qu'il fasse.

C'est pourquoi même les projets qui ne sont pas formellement organisés comme des démocraties s'en remettent dans la pratique à un processus démocratique quand il s'agit de prendre une décision importante. Pouvoir dupliquer le code implique pouvoir créer un fourche et la possibilité de créer un fourche implique le consensus. Il se peut très bien que tout le monde accepte de s'en remettre à un leader (l'exemple le plus célèbre étant Linus Torvalds et le développement du noyau Linux), mais parce qu'on choisit de le faire de manière totalement dénuée de cynisme et de malignité. Le dictateur n'a pas d'emprise magique sur le projet. Une des caractéristiques essentielles des licences Open Source est qu'elles mettent tout le monde sur un pied d'égalité face aux décisions concernant la modification ou l'utilisation du code. Si le dictateur se mettait soudain à prendre de mauvaises décisions, un mécontentement s'élèverait alors, suivi éventuellement d'une révolte et d'une fourche. En réalité, les choses vont rarement aussi loin, car le dictateur cède le premier.

La possibilité de fourche borne le pouvoir que chacun peut exercer au sein du projet, mais cela ne veut pas dire qu'il existe une façon unique de gouverner les projets. Il ne s'agit pas de brandir la menace de fourche à chaque décision, cette menace ne doit être utilisée qu'en dernier recours. Cela deviendrait vite fatigant et détournerait l'énergie du travail à réaliser. Les deux sections suivantes passent en revue différentes manières d'organiser les projets de sorte que les décisions soient prises en douceur. Les deux exemples cités sont des extrêmes un peu idéalisés ; la plupart des projets se situant quelque part entre les deux.

2. Les dictateurs bienveillants

Le dictateur bienveillant est exactement ce que laisse entendre son nom : le pouvoir de prise de décision ultime est mis entre les mains d'une personne dont on attend, en vertu de son expérience, qu'elle en use sagement.

Bien que « dictateur bienveillant » (DB) soit le terme courant pour ce rôle, il vaut mieux l'imaginer comme un « arbitre approuvé par la communauté » ou un « juge ». Généralement, les dictateurs bienveillants ne prennent pas concrètement toutes les décisions, ni même la plupart. Il est peu probable qu'une seule personne puisse avoir les compétences requises dans tous les domaines d'un projet et, de toutes façons, les bons développeurs ne s'éterniseront pas s'ils n'ont pas quelque influence sur la direction du projet. C'est pourquoi les dictateurs bienveillants ne dictent généralement pas grand-chose. En revanche, ils laissent les choses s'éclaircir d'elles-mêmes au cours de la discussion et de l'expérimentation, quand c'est possible. Ils y participent eux-mêmes, mais en tant que simples développeurs, s'en remettent souvent à un responsable du domaine ayant plus de savoir-faire. C'est seulement quand il apparaît clairement qu'un consensus ne peut être trouvé et que la majorité veut que quelqu'un prenne une décision afin que le développement puisse continuer, qu'il tape du poing sur la table en disant : « Voilà ce qu'il faut faire. ». Presque tous les dictateurs bienveillants ont en commun une aversion pour la prise de décisions par diktat ; c'est une des raisons pour lesquelles ils parviennent à garder leur rôle.

Qui peut être un bon dictateur bienveillant ?

Être un bon DB requiert une diversité de traits de caractère. Il faut, tout d'abord, une perception bien affûtée de sa propre influence sur le projet, ce qui implique faire preuve de modération. Dans les premières phases de discussion, si le DB assène ses opinions et ses conclusions avec force, les autres trouveront inutile de penser différemment. Les participants doivent se sentir à l'aise pour exposer leurs idées, y compris celles qui semblent les plus stupides. Inévitablement, le DB, lui aussi, postera à l'occasion une idée absurde, c'est pourquoi ce rôle demande aussi de savoir comprendre et reconnaître ses erreurs lors d'une mauvaise décision. C'est là un trait de caractère que tout bon développeur devrait posséder, notamment s'il reste longtemps dans le projet. La différence est que le DB peut se permettre des erreurs de temps en temps, sans craindre de conséquences sur sa crédibilité à long terme. Les développeurs de moindre ancienneté ne se sentiront pas aussi protégés, et le DB devra formuler ses critiques et décisions contraires avec courtoisie vu la portée de ses mots, aussi bien sur le plan technique que psychologique.

Le DB n'a nullement besoin de posséder les meilleures compétences techniques au sein du projet. Il doit en avoir suffisamment pour travailler sur le code, pour comprendre et discuter tout changement : c'est tout. La position de DB n'est ni obtenue ni détenue en vertu d'un talent de codeur hors du commun. Ce qui compte, c'est l'expérience et la vision globale, pas forcément la capacité à concevoir de bonnes choses à la demande, mais plutôt l'aptitude à reconnaître une bonne conception, quelle qu'en soit la source.

Le dictateur bienveillant est généralement un des fondateurs du projet, mais il s'agit plus d'une corrélation que d'une cause. L'ensemble des qualités permettant à une personne de

démarrer un projet avec succès (compétences techniques, capacité à persuader d'autres de s'y associer, etc.), est exactement identique aux aptitudes nécessaires au DB. Bien évidemment, les fondateurs débutent avec une sorte d'ancienneté automatique, souvent suffisante pour faire apparaître le dictateur bienveillant comme la solution la plus évidente s'offrant à tous.

Souvenez-vous que la potentialité de fourche est à double tranchant. Un DB peut scinder un projet aussi facilement que quiconque, et certains l'ont fait à l'occasion, quand ils ont senti que la direction voulue pour le projet était différente de celle souhaitée par la plupart des développeurs. Dans la mesure où une fourche est possible, peu importe que le dictateur bienveillant ait ou non les privilèges d'administrateur sur le serveur principal du projet. On parle parfois du contrôle du serveur comme d'une source ultime de pouvoir dans un projet, mais en fait c'est sans importance. La capacité, d'ajouter ou d'enlever, sur un serveur particulier, les mots de passe de validation (*commits*) des uns et des autres, n'affecte que la copie du projet hébergée sur ce serveur. En cas d'abus prolongé de ce pouvoir, de la part du DB ou d'un autre, il suffit de migrer le développement sur un autre serveur.

Quant à savoir si votre projet doit avoir un dictateur bienveillant ou s'il fonctionnerait mieux avec un système moins centralisé, cela dépend en grande partie du candidat disponible pour assumer ce rôle. En règle générale, s'il est évident, pour tout le monde, qu'untel devrait être le DB, le problème est résolu. En revanche, si aucun candidat ne se démarque nettement, le projet devrait plutôt recourir à un mode de prise de décision décentralisé.

3. La démocratie basée sur le consensus

En vieillissant, les projets tendent à s'éloigner du modèle du dictateur bienveillant au profit de systèmes plus ouvertement démocratiques. Ce n'est pas forcément par mécontentement à l'égard d'un DB en particulier. C'est simplement parce que la gouvernance basée sur le groupe est, en empruntant une métaphore à la biologie, plus « évolutionnairement stable ». Chaque recul, ou essai d'une répartition plus équilibrée de la responsabilité de décision, de la part du dictateur bienveillant, est une occasion pour le groupe d'établir un nouveau système non-dictatorial, de fonder une constitution en quelque sorte. Le groupe peut ne pas saisir cette opportunité la première fois, ni la deuxième, mais il finira par la prendre, et, une fois le processus engagé, un retour en arrière est peu probable. La raison relève du bon sens : si un groupe de N personnes devait investir un membre de pouvoirs spéciaux, ceci voudrait dire que N-1 personnes ont accepté de diminuer leur influence personnelle, ce qui est peu probable. Dans les faits, la dictature résultante n'est que conditionnelle : le DB sacré par le groupe peut, aussi bien, être détrôné par celui-ci. Ainsi, une fois qu'un projet a évolué d'une direction autocratique vers un système officiel basé sur le groupe, il revient rarement en arrière.

Le fonctionnement de ces systèmes connaît de nombreuses variantes, mais il existe 2 éléments communs : 1) la plupart du temps, le groupe travaille par consensus ; 2) il existe un mécanisme officiel de vote sur lequel se reposer quand le consensus n'est pas atteint.

« Consensus » veut simplement dire : accord prêt à être respecté par chacun. Il ne s'agit pas d'un état ambigu : le consensus est atteint, pour une question donnée, lorsqu'on le déclare

comme tel et que cette affirmation n'est pas contredite. La personne proposant le consensus devrait, bien sûr, préciser sur quoi il porte au juste, et quelles actions en découlent, si cela n'est pas évident.

La plupart des conversations, dans un projet, portent sur des questions techniques, telles que : la meilleure façon de corriger un bogue, l'ajout ou non d'une fonctionnalité, le degré de précision dans la documentation des interfaces, etc. La gouvernance basée sur le consensus fonctionne bien, car elle s'harmonise parfaitement à la discussion technique proprement dite. En fin de discussion, tout le monde s'accorde souvent sur la voie à prendre. Habituellement, une conclusion est postée faisant office conjointement de résumé des décisions et de proposition implicite de consensus. Ce qui offre une dernière chance à chacun de dire : « Attendez, je n'ai pas donné mon accord là-dessus. Il faut encore décortiquer la question. »

Pour les décisions mineures ne suscitant pas la controverse, la proposition de consensus est implicite. Par exemple, quand un développeur valide spontanément une correction de bogue, la validation elle-même (« *commit* ») contient une proposition de consensus : « Je suppose tout le monde d'accord, ce bogue doit être corrigé, et c'est la bonne façon de le faire. ». Bien sûr, le développeur ne le dit pas réellement ainsi, il ne fait que valider la correction, et les autres membres du projet ne se donnent pas la peine de manifester leur accord, car « qui ne dit mot consent ». Si une modification validée n'est, finalement, pas approuvée par tous, le projet engage alors simplement une discussion, comme si elle n'avait pas encore été validée. Le bien-fondé de cette méthode est l'objet de la prochaine partie.

La gestion de versions, gage de sérénité

Puisque l'on met le code source du projet sous gestion de versions, on peut revenir sur la plupart des décisions. Le cas le plus courant survient quand on valide un changement en pensant à tort satisfaire tout le monde, alors que les objections sont nombreuses. Il est alors d'usage de commencer obligatoirement par présenter ses excuses pour avoir manqué les discussions antérieures, du moins si l'on en trouve trace dans les archives de la liste. D'une manière ou d'une autre, il n'y a aucune raison que le ton de la discussion soit différent avant et après la validation du changement. Tout changement est réversible, du moins jusqu'à l'implantation des modifications de dépendances liées aux changements (c'est-à-dire, du code ajouté par la suite qui casserait si le changement original était soudain supprimé). Le logiciel de gestion de versions permet au projet de défaire les effets de jugements mauvais ou hâtifs. Il est donc possible de suivre librement son instinct, et d'évaluer le meilleur moment de l'ajout d' une modification en fonction des commentaires reçus.

Cela veut également dire que le processus pour parvenir au consensus n'a nul besoin d'être très formel. La plupart des projets le font au jugé. Les changements mineurs passent sans discussion, sinon minimale et ponctuée de quelques signes d'approbation. Concernant les changements plus importants, notamment ceux pouvant déstabiliser une bonne quantité de code, il faudrait attendre un jour ou deux avant de confirmer le consensus : la logique voudrait que personne ne devrait être tenu à l'écart d'une conversation importante, seulement pour n'avoir pas vérifié son courrier assez fréquemment.

Ainsi, une fois certain de savoir ce qu'il faut effectuer, on devrait simplement se lancer et le faire. Ceci s'applique non seulement aux correctifs logiciels, mais également aux mises

à jour du site Web, aux changements de la documentation et à tout ce qui a peu de chances de prêter à controverse. Généralement, les situations où il faut annuler une action sont rares, elles pourront donc être traitées au cas par cas. Bien sûr, on ne devrait pas encourager les autres à l'obstination. Il subsiste toujours une différence psychologique entre une décision en discussion et une décision déjà effective, même si elle est techniquement réversible. Ceux qui associent toujours l'élan à l'action seront moins disposés à défaire un changement qu'à l'empêcher en premier lieu. Cependant, si un développeur abuse en validant trop rapidement des changements potentiellement controversés, les autres sont en droit, et ont le devoir, de se plaindre et de lui imposer des règles plus strictes jusqu'à ce que la situation s'améliore.

Quand le consensus n'est pas possible, votez

Certains débats n'aboutissent pas à un consensus, c'est inévitable. Quand tous les autres moyens de sortir d'une impasse échouent, la solution est le vote. Mais avant d'y recourir les choix possibles doivent être clairement définis. Ici, de nouveau, le processus normal de la discussion technique se marie étonnamment avec les procédures de prise de décision du projet. Le genre des questions amenant au vote touche souvent à des thématiques complexes, aux multiples facettes. Dans de telles discussions alambiquées, une ou deux personnes, généralement, jouent le rôle de médiateurs : elles postent régulièrement des compte-rendus sur les différents arguments, et gardent trace des principaux points d'achoppement (et d'accord). Ces compte-rendus aident tout le monde à mesurer les progrès effectués et rappellent les questions en suspens. Ces mêmes compte-rendus peuvent servir de prototypes au bulletin de vote, au cas où ce dernier deviendrait nécessaire. Si les médiateurs ont bien fait leur travail, ils seront en mesure d'appeler à voter le moment venu, et le groupe sera disposé favorablement à l'utilisation de la feuille de scrutin basée sur leur compte-rendu. Les médiateurs eux-mêmes peuvent participer au débat et n'ont pas obligation de rester au-dessus de la mêlée pourvu qu'ils arrivent à comprendre et présenter équitablement les autres points de vue, sans que leur sentiment partisan les empêche de rendre compte de manière neutre de l'état de la discussion.

Le contenu du bulletin n'est, généralement, pas controversé. Quand l'affaire en vient au vote, quelques positions-clés bien identifiées sont déjà isolées et résumées par des descriptions succinctes. Occasionnellement, un développeur fera une objection sur le vote lui-même. Parfois son inquiétude est légitime, par exemple, si une option importante a été omise ou insuffisamment renseignée. Mais d'autres fois, un développeur essaiera tout simplement d'éviter l'inévitable, en sachant peut-être que l'issue du vote n'ira pas dans son sens. Référez-vous à la section « Les personnes difficiles » dans le chapitre 6 pour savoir comment gérer ce genre d'obstructionnisme.

Pensez à préciser le système de vote, vu qu'il en existe plusieurs, et que l'on pourrait se méprendre sur la procédure en cours. Dans la plupart des cas, la bonne option est le vote par approbation où chaque participant peut voter, comme il l'entend, selon ses choix sur le bulletin. Le vote par approbation est simple à expliquer et à compter, de plus, à la différence d'autres méthodes, c'est une élection à un seul tour. Reportez-vous à la page Wikipédia pour plus de détails sur le vote par approbation et les autres systèmes d'élection [1], mais évitez

1. http ://fr.wikipedia.org/wiki/Syst%C3%A8me_de_vote

d'entrer dans un long débat sur le choix (car, évidemment, vous vous trouverez lancé dans un débat sur le système de vote pour décider du système de vote lui-même !) Une des raisons qui font du vote par approbation un bon choix, est la difficulté d'y opposer des objections : c'est un système de vote presque aussi juste que possible.

Enfin, procédez au vote publiquement. Ni le secret, ni l'anonymat ne sont requis pour un vote portant sur des questions qui, de toute façon, ont été débattues en public. Faites en sorte que chaque participant poste son vote sur la liste de discussion du projet afin que n'importe quel observateur puisse compter et vérifier les résultats, et que tout soit consigné dans les archives.

Quand voter

Le plus difficile, en ce qui concerne le vote, est de déterminer quand il doit avoir lieu. En règle générale, le recours au vote ne devrait intervenir que lorsque toutes les autres options ont échoué. Ne voyez pas le vote comme LE moyen génial de clore les débats, loin de là. Il met fin à la discussion, et met donc fin à la réflexion créative sur le problème concerné. Tant que la discussion continue, subsiste la possibilité qu'une personne présente une nouvelle solution satisfaisant tout le monde. Étonnamment, ceci arrive assez souvent. D'un débat animé peut naître une nouvelle façon d'aborder le problème, laquelle débouche sur une proposition qui, finalement, satisfait tout le monde. Même si aucune proposition nouvelle ne surgit, il reste encore préférable de négocier un compromis plutôt que de voter. Après un compromis, tout le monde est un peu mécontent, tandis qu'après un vote, certains sont malheureux et d'autres heureux. D'un point de vue politique, la première situation est préférable : au moins chacun peut avoir le sentiment d'avoir tiré un prix de son malheur. On est peut-être malheureux, mais les autres le sont aussi.

L'avantage principal du vote est qu'il permet de régler définitivement une question afin de pouvoir aller de l'avant. Mais il règle la question au nombre de voix, et non pas par un dialogue rationnel amenant tout le monde à la même conclusion. Il me semble que ceux qui ont de l'expérience dans les projets Open Source sont moins enclins à régler les questions par le vote. Ils essaieront plutôt d'explorer des solutions n'ayant pas été examinées précédemment, ou feront des compromis plus élaborés que ceux initialement prévus.

Il existe plusieurs techniques pour éviter un vote prématuré. La plus évidente consiste simplement à dire : « Je pense que nous ne sommes pas encore prêts pour un vote », tout en exposant les raisons. Une autre consiste à demander un vote informel (non contraignant) à main levée. Si l'issue penche nettement en faveur d'une partie, certains seront alors plus enclins à négocier, évitant le recours à un vote formel. Mais le mieux reste encore d'offrir une nouvelle solution, ou un nouveau point de vue sur une proposition ancienne, ainsi les contributeurs se réinvestiront dans les problèmes au lieu de simplement ressasser leurs arguments.

Dans certains cas rares, tout le monde s'accorde sur le fait que les solutions avec compromis sont pires que celles sans arrangement. Alors, le vote devient plus acceptable, pour deux raisons : c'est le meilleur moyen d'amener une solution de qualité et, les participants ne seront pas foncièrement malheureux, quelle qu'en soit l'issue. Malgré tout, le vote ne doit

pas être précipité. La discussion conduisant au vote instruit l'électorat : interrompre trop tôt cette discussion nuirait donc à la qualité du résultat.

(Notez que ce conseil de ne pas s'obstiner à voter ne s'applique pas au vote touchant l'inclusion des modifications, décrit dans la section « Stabiliser une version » du chapitre 7. Là, le vote est bien plus un mécanisme de communication, un moyen d'enregistrer l'implication de chacun dans le processus de révision des modifications, afin de pouvoir évaluer l'attention dont a fait l'objet une modification).

Qui vote ?

Un système de vote c'est bien, mais qui a le droit de vote ? C'est là une question potentiellement sensible, car elle oblige le projet à distinguer certaines personnes selon leur implication ou leur jugement.

La meilleure solution consiste à utiliser une distinction existante, comme l'accès à la validation (« accès de *commit* »), et à y attacher les privilèges du vote. Dans les projets proposant des accès de *commit* à la fois restreints et non-restreints, les *committers* restreints doivent-ils aussi avoir le droit de vote ? La réponse à cette question dépend largement du processus accordant l'accès de *commit* restreint. Si le projet les délivre généreusement, par exemple comme un moyen de mettre à jour de nombreux outils versés au dépôt par des tierces parties, alors il faudrait préciser que l'accès de *commit* restreint est juste un droit de validation, pas un droit de vote. Et inversement : puisque ceux qui ont un accès de *commit* non-restreint auront les privilèges du vote, il faut les choisir non seulement en tant que programmeurs, mais aussi en tant que membres de l'électorat. Si quelqu'un affiche sur les listes des diffusion des tendances turbulentes ou obstructionnistes, le groupe devrait peut-être réfléchir à deux fois avant de lui donner les droits de *committer*, même si ses compétences techniques ne font aucun doute.

Le système de vote lui-même devrait être utilisé pour choisir les nouveaux *committers*, qu'ils aient un accès de *commit* restreint ou complet. Mais il s'agit ici d'une des circonstances où le vote secret est approprié. Il est impossible de voter pour un *committer* potentiel sur la liste de discussion publique sans risquer de heurter les sentiments du candidat (et sa réputation). En revanche, l'usage veut qu'un *committer* poste, sur une liste privée composée uniquement d' autres *committers*, la proposition d'accorder, à un tiers, l'accès à la validation. Les autres *committers* donnent ouvertement leur avis, sachant que la discussion est privée. Généralement, il n'y a aura pas de désaccord, donc pas besoin de vote. Après avoir attendu quelques jours afin d'être sûr que chacun a eu l'occasion de réagir, l'émetteur original de la proposition écrit au candidat pour lui offrir l'accès à la validation. Si désaccord il y a, une discussion s'ensuit, comme pour tous les autres désaccords, débouchant, le cas échéant, sur un vote. Pour que ce processus soit franc et ouvert, le simple fait que la discussion ait lieu devrait rester secret. Si la personne en question savait ce qui se trame, et ne se voyait pas offrir l'accès de *commit*, elle en conclurait qu'elle n'a pas obtenu l'adhésion, et en serait vraisemblablement blessée. Bien sûr, si quelqu'un demande explicitement l'accès à la validation, il ne reste plus alors qu'à considérer la proposition, et l'accepter ou la rejeter explicitement. Dans ce dernier cas, il faudrait procéder aussi poliment que possible, avec une explication claire : « Nous avons apprécié tes correctifs, mais nous avons besoin d'en

voir plus encore » ou « Nous avons apprécié tous tes correctifs, mais ils avaient besoin de beaucoup d'ajustements avant de pouvoir être appliqués, donc nous ne sommes pas prêts à te donner un accès à la validation pour le moment. Nous espérons que cela va changer avec le temps ». Souvenez-vous, ce que vous dites peut blesser, tout dépend du degré de confiance en elle de la personne. Essayez de voir les choses de son point de vue lorsque vous rédigez le courriel.

Ajouter un nouveau *committer* étant plus important que la plupart des autres décisions, certains projets ont des exigences particulières concernant le vote. Ils peuvent, par exemple, exiger que la proposition reçoive n votes positifs et aucun vote négatif, ou bien qu'une majorité la plébiscite. Les paramètres exacts importent peu : l'idée principale est que le groupe doit être prudent au moment d'ajouter de nouveaux *committers*. De telles exigences, similaires voire plus strictes, peuvent être appliquées au vote visant à retirer ses privilèges à un *committer* : en espérant que cela ne soit jamais nécessaire. Reportez-vous à la section intitulée « Committers » dans le chapitre 8 pour plus d'informations sur les options autres que le vote pour l'attribution ou le retrait des droits de *commit*.

Vote ou sondage ?

Pour certains votes, il peut être utile d'élargir l'électorat. Par exemple, quand les développeurs ne parviennent pas à déterminer si le choix d'une interface donnée correspond bien à l'usage réel du logiciel. Une solution peut être de demander à tous les abonnés de la liste de discussion du projet de voter. Il s'agit, en réalité, plus d'un sondage que d'un vote, et les développeurs peuvent ou non choisir de se conformer au résultat. Comme pour tout sondage, faites en sorte que les participants comprennent parfaitement qu'il s'agit de questions ouvertes : si quelqu'un propose une meilleure option que celles proposées dans le sondage, sa réponse pourrait être le résultat le plus important.

Veto

Quelques projets autorisent aussi un type particulier de vote : le veto. Le veto permet au développeur de figer un changement hâtif ou inconsidéré, au moins suffisamment longtemps pour que l'on puisse encore en discuter. Le veto se situe à mi-chemin entre une très forte objection et une forme d'obstruction. Son sens exact varie d'un projet à l'autre. Certains projets rendent très difficile l'annulation d'un veto, d'autres la permettent au moyen d'un vote majoritaire, peut-être après un délai obligatoire afin de prolonger la discussion. Tout veto se doit d'être accompagné d'une explication approfondie : sans elle, il devrait être considéré comme nul et non avenu.

Avec le veto vient le problème de son abus. Parfois les développeurs sont trop pressés de faire monter les enchères en lançant un veto quand le temps est encore aux discussions. Vous pouvez éviter l'abus de veto en étant vous-même très réticent à y recourir, et en signalant gentiment quand quelqu'un d'autre utilise son droit de veto trop souvent. Si nécessaire, vous pouvez également rappeler au groupe que les vetos ne sont contraignants que dans la mesure où le groupe accepte qu'ils le soient. Après tout, si une nette majorité de développeurs

veulent X, c'est X qui adviendra d'une façon ou d'une autre. Dans ce cas, soit le développeur opposant son veto cède, soit le groupe décide d'affaiblir la portée du veto.

Vous pourrez voir certains écrire « -1 » pour exprimer leur veto. Cet usage vient de l'Apache Software Foundation, dont le processus de vote [1] et de veto est hautement structuré. Les critères d'Apache se sont étendus à d'autres projets, vous retrouverez ses conventions utilisées à divers degrés dans de nombreux projets Open Source. Techniquement, « -1 » n'indique pas toujours un veto officiel, même d'après les critères d'Apache, en revanche, ce « -1 » est assimilé, de manière informelle, à un veto ou à une très forte objection.

Comme le vote, le veto peut être rétroactif. On ne peut faire objection au veto en prétextant que le changement en question a déjà été validé par « *commit* », ou que l'action a déjà été lancée (à moins qu'il ne s'agisse d'un fait irréversible, comme la publication d'un communiqué de presse). D'un autre côté, un veto arrivant des semaines ou des mois en retard a peu de chances, à juste titre, d'être pris très au sérieux.

4. Tout mettre par écrit

Arrivé à un certain point, le nombre de conventions et d'accords tacites entourant votre projet risque de devenir si important qu'il deviendra nécessaire de les consigner quelque part. Pour donner à ce document sa légitimité, faites savoir qu'il est basé sur les discussions menées sur la liste et sur des accords déjà en cours. Lors de sa rédaction, veillez à faire référence aux fils pertinents dans les archives de la liste de diffusion, et s'il existe des points sur lesquels vous avez des doutes, reposez la question. Le document ne doit contenir aucune surprise : ce n'est pas la source des accords, mais une simple description. Certes, s'il est réussi, on commencera à le citer comme référence, mais cela signifiera seulement qu'il reflète exactement la volonté générale du groupe.

La section intitulée « Les directives développeurs » dans le chapitre 2 fait allusion à ce document. Naturellement, si le projet est récent, il faut mettre par écrit les lignes directrices sans pouvoir s'appuyer sur les avantages d'une longue histoire. Mais au fur et à mesure que la communauté de développement mûrit, vous adapterez le discours afin de refléter l'évolution du projet.

N'essayez pas d'être exhaustif. Aucun document ne peut contenir tout ce qu'il est nécessaire de savoir pour participer à un projet. Parmi les conventions générées par un projet, un bon nombre restent souvent inexprimées, voire ne sont jamais mentionnées explicitement, et pourtant, tout le monde les respecte. D'autres encore sont tout bêtement trop évidentes pour être mentionnées, et ne feraient que détourner l'attention d'éléments importants moins évidents. Il est inutile, par exemple, d'écrire des directives telles que « Soyez poli et respectueux sur les listes de discussion, ne lancez pas de trolls » ou « Écrivez du bon code dépourvu de bogues ». Bien sûr, ces comportements sont souhaitables, mais ils sont si évidents qu'il est inutile de les mentionner. Si certains sont grossiers sur les listes, ou s'ils écrivent du code bogué, ils ne vont pas cesser de le faire simplement parce que c'est écrit dans les directives du projet. Ces situations doivent être traitées lorsqu'elles se présentent, et non par des incitations génériques. D'un autre côté, si le projet a des directives spécifiques afin d'écrire du

1. http ://www.apache.org/foundation/voting.html

code de qualité, telles que des règles pour documenter chaque API dans un certain format, alors elles devraient être détaillées par écrit.

En vous basant sur les questions les plus courantes des nouveaux venus ainsi que sur les récriminations les plus fréquentes des développeurs expérimentés, vous aurez une bonne idée des points à faire figurer dans ce document. Ceci ne veut pas forcément dire qu'il se présentera sous la forme d'une FAQ, il nécessitera vraisemblablement une structure narrative plus cohérente que ne le permet une FAQ. Mais il devrait suivre le même principe : traiter des questions posées réellement, et non celles, qui selon vous, pourraient se poser.

Si le projet est une dictature bienveillante ou si certains cadres sont investis de pouvoirs spéciaux (président, directeur ou autre), ce document est aussi une bonne occasion de codifier les procédures de succession. Parfois, cela se résume simplement à la désignation de personnes particulières pour remplacer le DB au cas où celui-ci quitterait brusquement le projet pour une raison quelconque. Généralement, si DB il y a, il est le seul pouvant nommer un successeur. S'il y a des cadres élus, alors, la procédure de nomination et d'élection ayant servi à les désigner à l'origine devrait être décrite dans ce document. Si aucune procédure n'existe, cherchez le consensus via la liste de discussion avant de la mettre par écrit. Les contributeurs sont parfois très chatouilleux en matière de hiérarchie, c'est pourquoi le sujet doit être abordé avec tact.

Mais le plus important est, peut-être, de dire clairement que les règles peuvent être révisées. Si les conventions décrites dans le document commencent à entraver le projet, rappelez bien à tout le monde que ledit document est censé être un portrait vivant des intentions du groupe et non une source de frustration et de blocage. Si quelqu'un prend l'habitude de demander, sans cesse, la révision des règles chaque fois qu'elles se mettent en travers de son chemin, vous n'êtes pas obligé d'en débattre en permanence : garder le silence est, parfois, une bonne tactique. Les autres personnes, en accord avec ses critiques, s'en mêleront, et le changement deviendra évident. Mais si personne n'est d'accord, le point soulevé ne suscitera aucune réaction, et le règlement restera tel quel.

Deux bons exemples de directives de projet sont : le Hacker's Guide [1] de Subversion et les documents de gouvernance de l'Apache Software Foundation (« How the ASF Works ? [2] » et « Consensus Gauging Through Voting [3] »). L'ASF est en réalité une collection de projets logiciels ayant un statut juridique d'organisme à but non lucratif, ces documents tendent donc à décrire des procédures de gouvernance plutôt que des conventions de développement. Mais leur lecture en vaut la peine car ils sont les fruits de l'expérience accumulée au cours de nombreux projets Open Source.

1. http ://svn.collab.net/repos/svn/trunk/www/hacking.html
2. http ://www.apache.org/foundation/how-it-works.html
3. http ://www.apache.org/foundation/how-it-works.html

L'argent

Ce chapitre aborde la question du financement d'un projet de logiciel libre. Il ne concerne pas uniquement les développeurs payés pour travailler sur des projets de logiciel libre, mais aussi leurs responsables, qui doivent comprendre les règles sociales régissant le cadre du développement. Dans les sections suivantes, nous posons l'hypothèse que le destinataire (« vous ») est soit développeur rémunéré, soit dirigeant. Les conseils dispensés aux deux catégories seront souvent identiques, sinon le contexte le précisera.

Le financement du développement d'un logiciel libre par une entreprise n'est pas un phénomène nouveau. Depuis toujours, nombreux sont les développements subventionnés de manière officieuse. Lorsqu'un administrateur système écrit un outil d'analyse du réseau pour l'aider dans son travail, puis, une fois en ligne, reçoit des corrections de bogues ou des contributions d'autres administrateurs système, il s'agit bien de la création d'un consortium officieux. Les entreprises qui rémunèrent ces administrateurs réseau et leurs fournissent bureau et bande passante en deviennent les mécènes involontaires. Elles profitent, bien sûr, de cet investissement,bien qu'elles n'en aient pas toujours conscience dans un premier temps.

La différence, aujourd'hui, est que ces collaborations se formalisent. Les entreprises ont compris les avantages des logiciels Open Source et commencent à s'impliquer plus directement dans leur développement. De même, les développeurs en sont arrivés à espérer que les projets vraiment importants attirent au moins des donations voire, si possible, des partenaires sur le long terme. Alors que la présence d'argent n'a rien changé aux principes de base du développement de logiciels libres, elle a profondément modifié la marche des choses, tant en terme d'effectifs que de temps par développeur. Cela a également eu une incidence sur l'organisation des projets, et sur les interactions entre les groupes impliqués. Les questions ne portent pas simplement sur la manière de dépenser l'argent, ou sur les moyens de mesurer le retour sur investissement. Elles touchent aussi à l'organisation et aux

procédures : comment des organisations hiérarchisées d'entreprises et de communautés du logiciel libre semi-décentralisées peuvent-elles travailler ensemble pour un meilleur profit ? S'accorderont-elles, d'ailleurs, sur le sens même de « meilleur profit » ?

Un soutien financier est, en général, bien accueilli par les communautés de développement Open Source. Il peut réduire la vulnérabilité du projet face aux forces du chaos qui emportent tant de projets avant qu'ils ne décollent vraiment, et il peut donc inciter les contributeurs à donner une chance au logiciel. Ils ont alors le sentiment d'investir leur temps dans quelque chose qui sera toujours présent six mois plus tard. Après tout, la crédibilité est particulièrement contagieuse. Quand IBM, par exemple, soutient un projet Open Source, on peut se dire que le projet n'aura pas le droit d'échouer. La confiance, inspirée par ce parrainage, se traduira en inspiration, un supplément de motivation qui peut transformer le projet en prophétie auto-réalisatrice.

Quoi qu'il en soit, le financement implique, également, une idée de contrôle. Manipulé sans attention, l'argent peut entraîner une scission entre développeurs rémunérés et bénévoles. Si ces derniers ont la sensation que c'est finalement l'argent qui décide de la conception ou des nouvelles fonctionnalités, ils partiront pour un projet paraissant plus proche d'une méritocratie que d'un travail à titre gracieux au bénéfice d'autres personnes. Ils ne s'en plaindront pas forcément ouvertement dans les listes de diffusion. Vous remarquerez plutôt que le nombre des contributions externes diminue à mesure que les volontaires cessent d'essayer vainement de se voir accorder une certaine crédibilité. Le bourdonnement de l'activité à petite échelle continuera, sous forme de rapports de bogues ou de petits correctifs occasionnels. Mais, vous ne verrez plus de grande contribution au code, ni de participation extérieure dans les discussions portant sur la conception. Les contributeurs ressentent ce que l'on attend d'eux, et répondent (ou non) à ces attentes.

L'argent n'est pas sans pouvoir pour autant. Vous pouvez influencer le projet par l'argent, mais pas de manière directe. Dans un échange commercial classique, vous donnez de l'argent en échange de ce que vous désirez. Si vous avez besoin d'une fonctionnalité supplémentaire, vous signez un contrat, payez votre part, et elle sera ajoutée. Dans un projet Open Source, ce n'est pas si simple. Vous pouvez signer un contrat avec quelques développeurs, mais ils vous mystifieraient s'ils garantissaient que la tâche sera acceptée par la communauté de développement simplement parce que vous l'avez payée. Le travail ne peut être accepté que pour son propre mérite, et s'il correspond aux projets de la communauté pour le logiciel. Vous pouvez avoir votre mot à dire dans ces projets, mais vous n'êtes pas le seul.

Vous n'achetez pas de l'influence, vous achetez une voix au sein du projet. Les programmeurs en sont le meilleur exemple. Si de bons programmeurs recrutés parviennent, avec le temps, à se faire respecter par la communauté, ils pourront influencer le projet à l'instar des autres membres. Ils auront le droit de vote et, s'ils sont nombreux, feront une alliance. Respectés dans le projet, leur influence dépassera leurs simples votes. De même, les développeurs rémunérés n'ont aucune raison de cacher leurs motivations. Après tout, quiconque veut une modification du logiciel la désire pour une raison. La motivation de votre entreprise est aussi légitime que toute autre. Il faut juste savoir que le poids accordé aux objectifs de votre entreprise sera déterminé par le statut de ses représentants au sein du projet, pas par sa taille ou son modèle économique.

1. Les différentes participations

Les projets Open Source peuvent être financés pour de nombreuses raisons. Les points de cette liste ne s'excluent pas mutuellement, souvent la décision de soutenir financièrement un projet résultera de plusieurs de ces motivations, voire de toutes :

Partager la charge — Des organismes distincts aux besoins logiciels communs finissent souvent par travailler en doublon, soit en écrivant un code proche et redondant en interne, soit en achetant des produits similaires à des vendeurs privés. Quand on comprend la situation, les structures peuvent fusionner leurs ressources et créer (ou rejoindre) un projet Open Source sur mesure satisfaisant les besoins. Les avantages sont évidents : coûts de développement partagés et gains accrus pour tout le monde. Ce scénario, plus logique pour des organisations à but non lucratif de prime abord, peut se révéler payant, même pour les entreprises commerciales. Exemples : openadapter.org [1], koha.org [2].

Accroître les services — Lorsqu'une entreprise vend des services dépendants de programmes Open Source particuliers, il est naturellement dans son intérêt de s'assurer que ces programmes sont activement maintenus. Exemple : Le soutien de CollabNet au site subversion.tigris.org [3] (NdA : c'est mon travail, mais c'est aussi un parfait exemple de ce modèle).

Renforcer l'offre matérielle — La valeur des ordinateurs et de leur composants est directement liée au nombre de logiciels disponibles. Les vendeurs de matériel (pas uniquement les vendeurs de machines complètes, mais aussi les fabricants de périphériques et de microprocesseurs) se sont aperçus que la disponibilité de logiciels libres de qualité fonctionnant sur leur matériel est importante aux yeux des clients.

Affaiblir un produit concurrent — Parfois, les entreprises apportent leur soutien à un projet Open Source particulier, dans le but d'affaiblir le produit d'un concurrent, Open Source ou non. Grignoter des parts de marché d'un concurrent, n'est, en général, pas la seule raison pour s'investir dans un projet Open Source, mais cela peut compter. Exemple : OpenOffice.org [4] (non, ce n'est pas la seule raison d'exister d'OpenOffice, mais le logiciel est au moins en partie une réponse à Microsoft Office).

Commercialisation — Associer son entreprise à une application Open Source populaire peut simplement être bon pour l'image de la société.

Double licence — Grâce à une double licence, vous pouvez proposer du logiciel, sous une licence privée classique, aux clients souhaitant le revendre au sein d'applications brevetées, mais aussi sous une licence libre, à ceux qui souhaitent l'utiliser comme logiciel Open Source (voir la section « La double licence » au chapitre 9). Si la communauté de développeurs Open Source est active, le logiciel bénéficie, à grande échelle, du développement et de la correction de bogues, mais la société perçoit

1. http ://www.openadapter.org/
2. http ://www.koha.org/
3. http ://subversion.tigris.org/
4. http ://www.openoffice.org/

toujours les redevances lui permettant de financer quelques programmeurs à temps plein. Deux exemples bien connus sont MySQL, les fabricants du logiciel de base de données du même nom, et Sleepycat qui s'occupe des distributions et du support de la Berkeley Database. Ce n'est pas une coïncidence si ces deux compagnies sont spécialisées dans les bases de données. Les logiciels de base de données sont plus souvent intégrés à d'autres applications que vendus directement aux utilisateurs, le modèle à licence double leur correspond donc très bien.

Les dons — Un programme très utilisé peut parfois recevoir des dons importants, de particuliers ou d'entreprises, en affichant simplement un bouton de donation en ligne, quelquefois en vendant des objets portant son logo comme des tasses à café, des T-shirts, des tapis de souris, etc. Attention : si votre projet accepte les dons, planifiez l'utilisation de cet argent avant qu'il n'arrive, et affichez vos prévisions sur le site Web du projet. Les discussions sur la manière d'allouer l'argent ont tendance à être plus tranquilles quand elles ont lieu avant qu'il n'y ait effectivement de l'argent à dépenser. De toutes façons, en cas de désaccords importants, mieux vaut s'en préoccuper quand tout n'est encore que théorique.

Le modèle économique d'un donateur n'est pas le seul facteur entrant en ligne de compte dans ses relations avec la communauté Open Source. L'historique de leurs rapports précédents est tout aussi important : l'entreprise est-elle à l'initiative du projet ou se joint-elle au développement pré-existant ? Dans les deux cas, le donateur devra mériter sa crédibilité, bien évidemment, le deuxième lui demandera plus d'efforts. L'organisme se doit d'avoir des buts précis pour le projet. L'entreprise essaie-t-elle de conserver une position dominante, d'être une simple voix parmi d'autres au sein de la communauté, ou, de guider mais sans forcément gouverner les décisions du projet ? Ou bien veut-elle simplement assurer ses intérêts en finançant quelques *committers* pour réparer les bogues rencontrés par ses clients et apporter des modifications dans la distribution publique sans faire de vagues ?

Gardez ces questions à l'esprit en lisant les conseils suivants. Ils sont faits pour être appliqués à tout investissement dans un projet de logiciel libre. En revanche, chaque projet étant un environnement humain, deux situations ne seront jamais identiques. Il faudra aussi vous fier à votre instinct. Cependant, en suivant ces conseils, vous devriez augmenter les probabilités d'un fonctionnement conforme à vos désirs.

2. Recrutez pour le long terme

Si vous dirigez des programmeurs dans un projet Open Source, faites en sorte qu'ils participent au projet assez longtemps pour acquérir l'expertise technique et politique nécessaire, deux ans au minimum. Évidemment, que le code source soit ouvert ou non, une rotation rapide des programmeurs n'est profitable à aucun projet. L'obligation d'apprendre tous les rouages, à chaque nouvel arrivant, serait dissuasive quel que soit l'environnement. Mais le handicap est encore plus fort pour les projets Open Source puisque les développeurs quittant le projet emmènent avec eux non seulement leur connaissance du code, mais aussi leur statut dans la communauté et les relations humaines qu'ils ont tissées.

La crédibilité acquise ne peut être transférée. Pour prendre l'exemple le plus évident, un nouveau développeur ne peut pas hériter de l'accès de *commit* du développeur sortant (voir

la section « L'argent ne fait pas tout » plus loin dans ce chapitre), donc, le nouveau développeur, n'ayant pas encore l'accès de *commit*, devra proposer des correctifs jusqu'à son obtention. Mais l'accès de *commit* n'est que la manifestation la plus visible de l'influence. Un développeur de longue date connaît toutes les discussions traitées et rabâchées sur les listes de discussion. Un nouveau développeur, ne connaissant pas toutes ces conversations, pourrait tenter de raviver un problème une nouvelle fois, affectant ainsi la crédibilité de votre organisation : les autres se diront « ne peuvent-ils donc pas s'en souvenir ? ». Un développeur novice ne connaîtra pas la hiérarchie interne du projet, et sera incapable d'influencer l'orientation du projet aussi rapidement et souplement qu'une personne présente depuis longtemps.

Formez les nouveaux venus au moyen d'un programme de recrutement supervisé. Le développeur inexpérimenté devrait être en contact direct avec la communauté de développement public dès le départ, en commençant par des corrections de bogues et des tâches de nettoyage afin d'apprendre les bases du code et se faire une réputation dans la communauté. Mais évitez de provoquer de longues discussions enflammées sur la conception du programme. Au cours du processus, un développeur expérimenté (ou plusieurs) devrait être disponible pour répondre à ses questions, il devrait aussi lire toutes les contributions faites par le nouvel arrivant à la liste de développement, même si elles se rapportent à des discussions habituellement ignorées par le développeur exercé. Cela aidera le groupe à repérer les écueils potentiels susceptibles de faire échouer le novice. Encouragements et indications en privé peuvent aussi aider grandement, notamment si le nouvel arrivant n'a pas l'habitude de voir son travail minutieusement inspecté par ses pairs.

Quand CollabNet engage un nouveau développeur pour travailler dans Subversion, nous choisissons ensemble quelques bogues, et le nouveau peut s'y faire les dents. Nous discutons du contour technique des solutions, ensuite, nous assignons au minimum un développeur expérimenté à la vérification (publique) du correctif que le nouveau développeur proposera publiquement. En règle générale, nous ne regardons même pas le correctif avant que la principale liste de développeurs n'en prenne connaissance, alors que nous pourrions le faire en cas de nécessité. Il est primordial que le développeur passe par le processus de révision publique, il apprend la base du code tout en s'habituant à recevoir des critiques de personnes complètement inconnues. Nous essayons aussi de coordonner nos réponses afin que nos impressions soient visibles dès la publication du correctif. Ainsi, les premières inspections lues sur la liste sont les nôtres, ce qui peut aider à donner la tonalité des révisions suivantes. Nous montrons ainsi, à tout le monde, que ce nouveau contributeur doit être pris au sérieux. Si l'on voit que nous prenons le temps de faire des commentaires détaillés, avec des explications exhaustives et des références aux archives si nécessaire, on comprend que le nouveau venu est dans une phase de formation, certainement synonyme d'investissement sur le long terme. Cela peut inciter tout le monde à être bien disposé envers ce développeur au point de consacrer du temps à ses questions et à inspecter ses correctifs.

3. Montrez-vous unis

Vos développeurs feront leur possible pour s'exprimer, sur les forums publics de développement, en tant que participants individuels plutôt que comme représentants d'une entreprise

monolithique. Ce n'est pas que la présence massive d'une entreprise ait une connotation négative (enfin, c'est possible, mais ce n'est pas le sujet de ce livre). C'est plutôt parce que les projets Open Source ne sont structurellement équipés que pour traiter avec des entités individuelles. Un participant individuel peut discuter, proposer des correctifs, gagner en crédibilité, voter, etc. Une entreprise ne le peut pas.

De plus, en décentralisant le projet, vous évitez la concentration de l'opposition. Laissez vos développeurs argumenter entre eux dans les listes de diffusion. Encouragez-les à réviser leur code mutuellement, aussi souvent que possible et publiquement, comme ils le feraient pour toute autre personne. Dissuadez-les de voter en groupe, car s'ils le font, d'autres pourraient commencer à penser, juste par principe, qu'ils devraient eux aussi s'organiser pour garder un certain poids.

Il y a une différence entre être vraiment décentralisé et s'efforcer de paraître décentralisé. Selon les circonstances, il peut s'avérer utile que vos développeurs agissent de concert, ils devraient donc être préparés à s'unir en coulisses si nécessaire. Par exemple, quand vous faites une proposition, l'intervention de quelques personnes, exprimant leur consentement, peut aider en donnant l'impression qu'un consensus s'installe. Les autres auront alors l'impression que la proposition gagne de la vitesse, et que faire part de leur objection casserait cet élan. En conséquence, ceux qui protestent ne le feront qu'à juste raison. Il n'y a aucun mal à orchestrer un accord de cette manière, tant que les objections sont prises au sérieux. Les expressions publiques d'un accord privé ne sont pas moins sincères parce qu'il a été conclu par avance, et ne sont dommageables que si elles sont utilisées afin d'étouffer les arguments contraires. Leur but est tout simplement d'inhiber cette catégorie de personnes qui ne contredit que pour la forme (voir la section nommée « Plus un sujet est facile plus les discussions sont longues » dans le chapitre 6 pour en savoir plus à ce propos).

4. Ne cachez pas vos motivations

Annoncez les buts de votre organisation en toute transparence, sans révéler de secrets commerciaux. Si vous voulez que le projet incorpore une certaine fonctionnalité parce que, par exemple, vos clients la réclament à cor et à cri, annoncez-le clairement sur les listes de diffusion. Si les clients souhaitent rester anonymes, comme c'est parfois le cas, demandez-leur au moins si vous pouvez les citer comme exemples sans les nommer. Les développeurs accepteront d'autant mieux vos propositions qu'ils en connaîtront les raisons.

Cela va à l'encontre de l'idée instinctive, si facile à acquérir et si difficile à chasser, que la connaissance est un pouvoir et qu'en jouant carte sur table, vous vous mettez en position de faiblesse. Mais cette idée est infondée ici. En défendant publiquement les fonctionnalités (ou correctifs de bogues ou quoi que ce soit d'autre), vous avez déjà dévoilé votre jeu. La question en suspens est de savoir si vous arriverez à mener la communauté à partager vos objectifs. Si vous dites simplement ce que vous désirez, mais sans fournir d'exemple, votre défense est faible, on commencera à soupçonner des motivations cachées. En revanche, donnez des exemples concrets, montrez pourquoi la fonctionnalité proposée est importante, et vous pourrez influer sur le débat de façon significative.

Pour comprendre cela, pensez à la situation inverse. Trop souvent, les débats à propos de nouvelles fonctionnalités ou de nouvelles orientations sont longs et fatigants. Les arguments

avancés se résument en général à « Moi je veux ceci ou cela », ou encore, à cette phrase extrêmement populaire « D'après mes années d'expérience en tant qu'architecte logiciel, ceci ou cela est important pour les utilisateurs / n'est qu'une fonction inutile qui ne plaira à personne ». Comme prévu, l'absence de données concrètes n'aide ni à écourter, ni à tempérer de tels débats. Au contraire, cela les éloigne toujours plus de la question centrale : la satisfaction de l'utilisateur. Sans la force d'un contrepoids, il est très probable finalement que le résultat ne sera pas déterminé par celui qui aura montré le plus de logique, de ténacité ou d'expérience.

En tant qu'entreprise possédant un grand nombre de données clients, vous avez la possibilité d'exercer cette force pour faire contrepoids. Vous êtes le fil conducteur transmettant les informations qui, autrement, n'atteindraient pas la communauté de développeurs. Utiliser ces informations pour justifier vos souhaits n'a rien d'embarrassant. La plupart des développeurs n'ont pas, individuellement, une grande expérience de l'utilisation qui est faite du logiciel qu'ils écrivent. Chaque développeur utilise le logiciel d'une manière qui lui est propre, et vous apportez à la communauté de développement public une vraie bouffée d'oxygène. Tant que vous y mettez les formes, ils l'accueilleront avec enthousiasme, et cela propulsera les choses dans la direction que vous souhaitez.

Tout réside, évidemment, dans la manière de présenter les choses. Vous n'obtiendrez jamais rien en insistant juste parce que vous représentez un grand nombre d'utilisateurs, et que, parce qu'ils ont besoin d'une fonctionnalité particulière (ou pensent en avoir besoin), votre solution devrait être ajoutée. Vous devriez, de préférence, concentrer vos premiers messages sur le problème, plutôt que sur une solution particulière. Décrivez avec force de détails les difficultés que rencontrent vos clients, donnez autant d'analyses et de solutions viables que possible. Quand les acteurs du projet commencent à spéculer sur l'efficacité de plusieurs solutions, vous pouvez continuer à déployer vos arguments pour soutenir ou réfuter leurs propos. Vous aviez peut-être déjà une solution depuis le début, mais ne la mettez pas tout de suite en avant. Ce n'est pas une tromperie, c'est simplement l'attitude standard du « courtier honnête ». Après tout, votre véritable but est de régler le problème, une solution n'est qu'un moyen d'arriver à cette fin. Si la solution que vous privilégiez est effectivement meilleure, d'autres développeurs finiront par le remarquer d'eux-mêmes et s'y rangeront de leur propre gré, ce qui vaut bien mieux que de les intimider pour qu'ils l'ajoutent (la probabilité qu'ils proposent une meilleure solution n'est pas nulle non plus).

Cela ne veut pas dire que vous devez vous interdire de prendre position pour une solution particulière. Mais vous devez avoir la patience de voir l'analyse, déjà menée personnellement, être répétée sur les listes de développement publiques. Ne répondez pas :« Oui, on y a déjà réfléchi, mais ça ne marche pas pour les raisons A, B et C. À bien y réfléchir, la seule solution pour régler ce problème est… ». Ce n'est pas tant l'arrogance qui pourrait transparaître, que l'impression d'y avoir déjà consacré en coulisses une quantité inconnue (qu'on présumera importante) de ressources analytiques qui pose problème. Les autres auront l'impression que des travaux majeurs ont déjà été réalisés, que peut-être des décisions ont été prises, et ce, sans concertation du public : c'est bien la meilleure voie vers la rancœur.

Naturellement, vous connaissez la quantité d'efforts dédiée en interne, et ce savoir est, d'une certaine manière, un désavantage. Vos développeurs se retrouvent dans un état d'esprit légèrement différent de celui des autres participants de la liste de diffusion. De fait, cela réduit leur capacité à appréhender les choses du point de vue des personnes n'ayant pas

encore beaucoup réfléchi au problème. Vous réduirez ce fossé en réussissant à rallier tout le monde à votre vision le plus tôt possible. Cette logique s'applique, non seulement à des problèmes techniques isolés, mais aussi à la tâche plus générale consistant à exposer aussi clairement que possible vos objectifs. L'inconnu fait toujours plus peur que le connu. Si les autres comprennent pourquoi vous voulez ce que vous voulez, ils se sentiront plus à l'aise pour vous parler, même si c'est pour exprimer leur désaccord. S'ils ne peuvent pas cerner ce qui vous motive, ils s'imagineront le pire, au moins de temps en temps.

Évidemment, il vous est impossible de tout rendre public vous-même, et personne n'attend que vous le fassiez. Toutes les entreprises ont leurs secrets, celles à but lucratif en ont peut-être plus, mais les organisations à but non-lucratif en ont aussi. Si vous devez défendre une certaine orientation, mais que vous ne pouvez rien dévoiler de vos raisons, donnez les meilleurs arguments que vous ayez malgré ce handicap, et acceptez le fait que vous n'aurez peut-être pas autant d'influence que vous le souhaiteriez dans la discussion. C'est l'un des compromis à faire pour avoir une communauté de développement absente du registre des salaires.

5. L'argent ne fait pas tout

Si vous êtes un développeur rémunéré au sein d'un projet, définissez rapidement ce que l'argent peut acheter ou non. Cela ne veut pas dire que vous devez envoyer deux messages par jour sur les listes de diffusion pour réaffirmer votre nature noble et incorruptible. Non, vous devez essentiellement montrer que vous êtes conscient que des tensions peuvent survenir à cause de l'argent. Inutile de présupposer dès le lancement du projet que ces tensions existent, mais faites savoir clairement que vous êtes prêt à désamorcer ces situations si elles venaient à se présenter.

Le projet Subversion me fournit un très bon exemple. Il a été lancé en 2000 par CollabNet, donateur principal dès sa création et employeur de plusieurs développeurs (NdA : je n'en fais pas partie). Peu après le début du projet, nous avons engagé un autre développeur, Mike Pilato, pour se joindre à nos efforts. À cette période, l'écriture du code avait déjà commencé. Bien que Subversion n'en fût encore qu'à ses balbutiements, il existait déjà une communauté de développeurs qui respectait un ensemble de règles de base.

L'arrivée de Mike a soulevé une question intéressante. Subversion avait déjà une règle pour l'attribution à un nouveau développeur de l'accès de *commit* : il commence par proposer des correctifs à la liste de développement. Lorsqu'il a fait ses preuves et que les autres *committers* voient que le nouvel arrivant sait ce qu'il fait, quelqu'un lui propose de valider ses changements lui-même (cette proposition est privée, comme décrit dans la section « Committers »). En supposant que tous les *committers* soient d'accord, l'un d'entre eux envoie un courrier au nouveau développeur et lui propose un accès de *commit* direct aux dépôts du projet.

CollabNet a engagé Mike tout spécialement pour travailler sur Subversion. Pour ceux qui le connaissaient déjà, ses aptitudes à coder et son envie de travailler sur le projet ne faisaient aucun doute. De plus, les développeurs volontaires avaient une très bonne relation avec les employés de CollabNet. Ils n'auraient certainement pas émis d'objection si nous

avions donné l'accès de *commit* à Mike dès le jour de son embauche. Mais nous savions que nous établirions alors un précédent. Si nous avions accordé l'accès à Mike sans passer par le processus habituel, nous aurions envoyé le message selon lequel CollabNet, par son statut de mécène principal, avait le droit d'ignorer les règles du projet. Même si les conséquences n'auraient pas forcément été visibles sur-le-champ, progressivement les développeurs bénévoles se seraient sentis privés de leur droit de vote. Les autres développeurs doivent mériter leur accès de *commit*, CollabNet l'achète.

Donc Mike accepta de commencer chez CollabNet comme tout autre développeur volontaire, sans accès de *commit*. Il envoya des correctifs aux listes de diffusion publiques, là où ils pouvaient être vérifiés par tout le monde (et ils l'ont été). Il fut aussi dit sur les listes que nous le faisions délibérément, afin que tout le monde comprenne bien la démarche. Après quelques semaines d'activité intense de la part de Mike, quelqu'un (je ne me souviens plus si c'était un développeur de CollabNet ou non) proposa qu'on lui donne l'accès de *commit*, lequel lui fut accordé, comme nous nous y attendions.

Cette cohérence vous apportera une crédibilité que l'argent ne pourra jamais acheter. Et la crédibilité est une monnaie précieuse dans les discussions techniques : elle fournit une immunité contre la remise en cause des motivations. Dans le feu de la discussion, les protagonistes ne se contenteront pas nécessairement d'arguments techniques pour remporter la bataille. Le principal donateur du projet, en raison de sa profonde implication et de son intérêt évident pour la direction prise par le projet, est une cible plus facile que les autres. En respectant scrupuleusement les règles du projet dès le début, le donateur ne s'expose pas plus que les autres.

Vous pouvez aussi lire une histoire similaire [1] racontée par D. Cooper sur Divablog à propos de l'accès de *commit*. Cooper était alors la « Diva Open Source » chez Sun Microsystem, je crois que c'était son titre officiel. Dans ce billet, elle décrit comment la communauté de développement Tomcat parvint à faire respecter par Sun les mêmes règles concernant l'accès de *commit* pour ses propres développeurs et les développeurs extérieurs.

Tout ceci implique également que le modèle de gouvernance du « Dictateur bienveillant » (voir la section « Le dictateur bienveillant » dans le chapitre 4) est plus compliqué à mettre en place quand il existe un mécénat, en particulier si le dictateur travaille pour le donateur principal. Une dictature ayant peu de règles, il est difficile pour le donateur de prouver qu'il se conforme aux standards de la communauté, même si c'est effectivement le cas. Ce n'est évidemment pas impossible, il suffit que le meneur du projet soit capable de voir les choses à la fois du point de vue d'un développeur extérieur et de celui du donateur. En tous cas, je vous conseille de garder sous le coude une proposition de système de gouvernance non dictatoriale, prête à être sortie dès que vous percevez les signes indicateurs d'une insatisfaction grandissante dans la communauté.

6. Signer des contrats

Le travail sous contrat doit être géré avec précaution dans les projets de logiciel libre. Idéalement, vous voulez que le travail de la personne sous contrat soit accepté par la communauté, et ajouté à la distribution publique. En théorie, peu importe qui est le signataire, tant

1. http ://blogs.sun.com/roller/page/DaneseCooper/20040916

que son travail est bon et satisfait aux lignes directrices du projet. La théorie et la pratique peuvent se rejoindre également : une personne complètement étrangère au projet proposant un bon correctif le verra en général ajouté au logiciel. Cependant produire un bon correctif, une amélioration importante ou une nouvelle fonctionnalité, pour une personne complètement étrangère au projet, n'a rien d'évident : elle doit d'abord en discuter avec le reste de l'équipe. La durée de cette discussion est par nature imprévisible. Un contrat payé à l'heure pourrait vous coûter nettement plus cher que prévu. Payé au forfait, le développeur pourrait se retrouver avec plus de travail qu'il ne peut en assurer.

Deux solutions existent. La plus courante est de prévoir intelligemment la durée du processus de discussion, en se fondant sur des expériences passées, avec une marge d'erreur, et baser le contrat là-dessus. Cela aide également à décomposer le problème en autant de parties indépendantes que possible, pour faciliter la prévision de chaque partie. L'autre possibilité est de signer le contrat uniquement pour la livraison d'un correctif, et de traiter séparément l'acceptation du correctif par le projet. Il devient alors beaucoup plus simple de rédiger le contrat, mais il vous incombe de maintenir le correctif tant que vous dépendez du logiciel, du moins le temps de parvenir à faire entrer le correctif, ou une fonctionnalité équivalente, dans le développement. Évidemment, même avec ces solutions privilégiées, le contrat ne peut pas imposer l'intégration du correctif au code, cela reviendrait à vendre quelque chose qui n'est pas à vendre (que se passerait-il si le reste du projet décidait sans prévenir de ne pas prendre en charge cette fonction ?). Cependant, le contrat peut exiger un effort de bonne foi pour que la modification soit acceptée par la communauté, et qu'elle soit validée dans le dépôt si la communauté le juge bon. Par exemple, si des règles existent au sujet des modifications du code, le contrat peut y faire référence, et préciser que le travail doit y être conforme. En pratique, tout fonctionne généralement comme prévu.

La meilleure tactique pour que la signature d'un contrat se révèle payante est d'engager un développeur du projet, un *committer* de préférence. L'influence d'un développeur au sein d'un projet est principalement due à la qualité du code qu'il produit et aux relations avec ses pairs. Le fait qu'il soit sous contrat pour mener à bien certaines tâches ne change en rien son statut, quoique cela puisse amener un examen plus attentif de ses faits et gestes. La plupart des développeurs ne risqueraient pas leur statut à long terme dans un projet en soutenant une nouvelle fonctionnalité inappropriée ou largement désapprouvée. En fait, en engageant un développeur, en plus du travail sur le code, vous obtiendrez (ou devriez obtenir) aussi des conseils concernant des modifications ayant une chance d'être acceptées par la communauté. Vous bénéficierez également d'un léger glissement des priorités du projet. Puisque les priorités sont établies en fonction des disponibilités de chacun, quand vous achetez le temps de quelqu'un, le travail de cette personne remonte un peu dans la liste des priorités. C'est une vérité bien comprise par les développeurs Open Source expérimentés, et certains d'entre eux feront attention au travail de la personne sous contrat, simplement parce qu'il a une chance d'être achevé : ils veulent donc aider à ce qu'il soit bien fait. Peut-être n'écriront-ils aucune ligne du code, mais ils discuteront tout de même de sa conception et en feront des vérifications, les deux pouvant être très utiles. Pour toutes ces raisons, il vaut mieux faire signer une personne parmi celles déjà impliquées dans le projet.

Deux questions surviennent alors : les contrats devraient-ils toujours être privés ? Et s'ils ne le sont pas, devez-vous vous inquiéter des possibles tensions engendrées par l'embauche de tel développeur plutôt que tel autre ?

Mieux vaut être transparent au sujet des contrats, quand vous le pouvez. Sinon l'attitude du signataire pourrait sembler étrange aux autres membres de la communauté : il va peut-être, soudain, vouloir donner une priorité importante à des fonctions auxquelles il n'avait jamais prêté attention par le passé. Quand on lui demandera la raison de ce revirement, comment pourra-t-il répondre de façon convaincante s'il ne peut évoquer le fait d'avoir signé un contrat pour les écrire ?

Ne soyez pas présomptueux non plus, ne faites pas passer (ni vous ni le contractant) vos accords pour plus importants qu'ils ne le sont. Trop souvent, j'ai vu des prestataires débarquer sur une liste de développement en pensant que leurs contributions allaient être prises plus sérieusement pour la seule raison qu'elles étaient payées. Ce genre d'attitude montre au reste du projet que pour la personne signataire, le contrat, et non le code résultant du contrat, est la chose qui compte. Mais pour les autres développeurs, seul le code compte. Quoi qu'il arrive, l'attention devrait être centrée sur les problèmes techniques, pas sur des histoires de qui paie qui. Un développeur de la communauté Subversion, par exemple, manie l'aspect contrat d'une manière particulièrement élégante. Pendant qu'il parle de ses modifications du code sur IRC, il dira à part (souvent sous la forme d'une remarque privée à l'un des *committers*) qu'il est payé pour son travail sur cette fonction, ou sur ce correctif en particulier. Mais en même temps, il donne systématiquement l'impression de vouloir de toute façon travailler sur ce changement et qu'il est content que l'argent lui en donne la possibilité. Il révèlera ou non l'identité de son client, mais dans tous les cas, il ne se repose pas sur le contrat. L'essentiel de son discours est technique, et porte sur la manière de mener les choses à bien, les mentions de ses contrats ne sont que des remarques.

Cet exemple illustre une autre raison de ne pas cacher l'existence des contrats. Il se peut que plusieurs organisations financent des contrats sur un même projet Open Source. Si toutes savent ce que les autres tentent de faire, elles peuvent partager leurs ressources. Dans le cas ci-dessus, le donateur principal du projet (CollabNet) n'est impliqué en aucune manière dans ces contrats ponctuels, mais savoir que quelqu'un d'autre finance certaines corrections de bogues permet à CollabNet de concentrer ses ressources sur d'autres bogues, ce qui au final améliore l'efficacité globale du projet.

Quelles conséquences aura cette barrière artificielle créée par les contrats ? En général, aucune, notamment quand les personnes rémunérées sont des membres très investis dans le projet et respectés par la communauté. Personne ne s'attend à ce que les contrats soient répartis équitablement entre tous les *committers*. Chacun comprend l'importance des relations à long terme : les incertitudes liées à la signature de contrats sont telles qu'une fois trouvée la personne avec qui vous pouvez travailler en toute confiance, vous êtes peu enclin à changer d'interlocuteur par simple souci d'équité. Voyez les choses ainsi : à la signature du premier contrat, il n'y aura pas de plainte puisqu'il fallait de toute évidence choisir quelqu'un (ce n'est pas de votre faute si vous ne pouvez engager tout le monde). Par la suite, engager la même personne une deuxième fois relève du bon sens : vous la connaissez déjà, votre association a fonctionné précédemment, alors pourquoi prendre des risques inutiles ? Il est donc parfaitement normal de se reposer sur une ou deux personnes dans la communauté plutôt que de répartir le travail de façon plus équilibrée.

Révision et acceptation des changements

Malgré tout, l'aboutissement d'un contrat dépend beaucoup de la communauté. L'implication des autres membres, dans la conception et le processus de vérification des modifications importantes, ne peut pas être ignorée. Elle doit être considérée comme une partie du travail, et être entièrement prise en compte par la personne sous contrat. Ne voyez pas l'examen fait par la communauté du projet comme un obstacle à surmonter, mais plutôt comme une contribution à la conception libre et à l'assurance qualité. Ce n'est pas une épreuve à endurer, c'est une opportunité à saisir.

Cas d'étude : le protocole d'authentification de mot de passe CVS

En 1995, nous n'étions qu'un binôme à assurer le support et les améliorations pour CVS (Concurrent Versions System [1]). Mon partenaire Jim et moi-même étions, de manière officieuse, les personnes chargées de maintenir CVS à ce moment. Mais nous n'avons jamais vraiment pris la peine de penser aux relations que nous entretenions avec la communauté de développement existante de CVS, des volontaires pour la plupart. Pour nous, ils envoyaient les correctifs et nous les appliquions, c'était à peu près comme ça que ça marchait.

À cette époque, le travail en réseau sur CVS n'était possible que grâce à un programme de connexion distant tel que *rsh*. Le fait d'utiliser le même mot de passe, pour l'accès à CVS et pour ce programme, était un risque de sécurité évident qui découragea de nombreuses entreprises. Une grande banque d'investissement nous engagea pour créer un nouveau système d'authentification, afin de pouvoir utiliser CVS en ligne de manière sécurisée depuis leur bureau distant.

Jim et moi avons accepté le contrat et pris le temps d'établir un nouveau système d'authentification. Notre solution était plutôt simple (les États-Unis exerçaient un contrôle sur l'exportation de code chiffré à cette époque, donc nos clients comprenaient que nous ne pouvions introduire une authentification forte), mais comme nous n'avions pas d'expérience dans ce domaine, nous avons commis quelques gaffes qui n'auraient pas échappé à un expert. Nous aurions facilement détecté ces erreurs si nous avions pris le temps d'écrire un premier jet et de le soumettre aux autres développeurs pour inspection. Mais nous ne l'avons jamais fait, ignorant les compétences des volontaires prêts à nous aider. Certains que tout ce que nous pourrions proposer serait sûrement accepté et ignorant nos lacunes, nous n'avons pas pris la peine de faire le travail de manière ouverte, c'est-à-dire, d'envoyer fréquemment des correctifs, en faisant des petites modifications, facilement digestibles pour une branche spécifique, etc. Le protocole d'authentification résultant n'était pas très bon, et bien sûr, une fois mis en place, difficilement améliorable pour des raisons de compatibilité.

L'expérience ne posait pas de problème, nous aurions facilement pu apprendre ce que nous devions savoir. C'est plutôt notre attitude vis-à-vis de la communauté de développeurs bénévoles qui posait problème. Nous considérions alors la validation des modifications comme un obstacle plutôt qu'une opportunité d'amélioration. Puisque nous savions que tout ce que nous faisions serait accepté (comme c'était effectivement le cas), nous n'avons pas fait l'effort d'impliquer les autres.

1. http ://www.cvshome.org/

De toute évidence, lorsque vous choisissez un contractant, vous souhaitez que cette personne ai les bonnes aptitudes techniques et une bonne expérience de la tâche visée. Mais il est aussi important de choisir quelqu'un dont l'expérience prouve qu'il a des relations constructives avec les autres développeurs au sein de la communauté. Ainsi vous n'engagez pas seulement une personne, vous engagez un agent susceptible de s'appuyer sur un réseau de compétences pour s'assurer que le travail réalisé est solide et facile à maintenir.

7. Financer ce qui ne touche pas à la programmation

Mais la programmation n'est la seule activité au sein d'un projet Open Source, loin s'en faut. Du point de vue des volontaires du projet, c'est la partie la plus visible et la plus glamour. Ce qui veut malheureusement dire que les autres activités, comme la documentation ou les tests formels par exemple, peuvent parfois être négligés, comparés à l'attention qu'ils reçoivent dans les logiciels propriétaires en tout cas. Les entreprises arrivent parfois à compenser cette lacune en assignant une partie de leur infrastructure de développement de logiciel interne à des projets Open Source.

La clé du succès de cette opération est de réussir à faire la transition entre les processus internes de l'entreprise et ceux de la communauté de développement publique. Une telle transition ne se fait pas sans heurts : souvent les deux groupes ne se ressemblent en rien, et les différences ne peuvent être gommées que par une intervention humaine. L'entreprise peut, par exemple, utiliser un système de suivi de bogues différent de celui du projet public. Et même en utilisant le même logiciel de suivi, les données qui y sont stockées seront très différentes puisque les besoins de suivi de bogues d'une entreprise ne sont vraisemblablement pas les mêmes que ceux d'une communauté de logiciel libre. Une information inscrite dans un suivi devrait peut-être apparaître dans l'autre, en enlevant les parties confidentielles ou, à l'inverse, en en ajoutant.

Les parties qui suivent traitent de la création et du maintien de ces connexions. En fin de compte, le projet Open Source devrait tourner plus souplement, la communauté reconnaissant le fait que l'entreprise investit une partie de ses ressources, sans pour autant avoir l'impression qu'elle poursuit uniquement ses propres objectifs.

Assurance qualité (ou : tests professionnels)

Les logiciels propriétaires bénéficient souvent de l'attention d'équipes dédiées entièrement à l'assurance qualité : chasse aux bogues, tests de performances et d'extensibilité, vérification de l'interface et de la documentation, etc. Par expérience, ces activités ne sont pas traitées avec autant de rigueur par la communauté de volontaires dans un projet de logiciel libre. C'est en partie dû au fait que les volontaires sont rares pour ces travaux peu valorisants, mais aussi parce que les développeurs comptent sur leur grande communauté d'utilisateurs pour avoir de nombreux retours d'expérience et, en ce qui concerne les performances et l'extensibilité, en partie parce que les volontaires ont, de toute façon, rarement accès aux ressources matérielles suffisantes.

Pourtant, l'équation « beaucoup d'utilisateurs = beaucoup de testeurs » n'est pas totalement dénuée de sens. Ce n'est certainement pas très utile d'assigner des testeurs aux

fonctions basiques en utilisation normale : ces bogues-là seront rapidement détectés par les utilisateurs au cours de leur usage courant du logiciel. Mais parce que les utilisateurs essayent simplement d'obtenir un résultat, ils ne vont pas d'eux-mêmes se mettre à explorer les confins inexplorés des fonctionnalités du programme, et laisseront certainement passer certaines catégories de bogues. De plus, quand ils trouvent une astuce pour contourner facilement un bogue, ils l'utilisent souvent sans prendre la peine de faire remonter le bogue. Plus insidieusement, l'utilisation du programme par vos clients (les personnes qui motivent votre intérêt pour le logiciel) peut être complètement différente de l'usage qu'en fait l'utilisateur lambda.

Une équipe de test professionnelle peut mettre à jour ce genre de bogues et ne fait pas de distinction entre un logiciel libre et un logiciel propriétaire. Le plus difficile reste de faire le lien entre l'équipe de test et l'équipe de développement. Les services de tests internes ont en général leur manière bien particulière de rapporter les résultats des tests, en employant un jargon spécifique à l'entreprise ou en s'appuyant sur une connaissance spécialisée de clients particuliers et de leurs données. Ces rapports ne sont pas adaptés à un système de suivi de bogues public, à la fois à cause de leur forme et pour des raisons de confidentialité. Même si le logiciel de suivi interne de bogues de votre entreprise est le même que celui utilisé par le projet public, les dirigeants pourraient avoir besoin de faire des commentaires spécifiques aux entreprises et d'apporter des changements aux métadonnées concernant les incidents (par exemple, pour élever la priorité interne du problème ou prévoir sa résolution pour un client en particulier). En général ce type de notes est confidentiel, parfois elles ne sont même pas montrées au client. Mais, quand bien même ne seraient-elles pas confidentielles, elles ne concernent en rien le projet public, et par conséquent, elle ne devraient pas distraire les volontaires.

C'est le cœur du rapport de bogue lui-même qui est important pour le public. En fait, un rapport de bogue émanant de votre service de tests a, en quelque sorte, plus de valeur qu'un autre envoyé par un utilisateur quelconque puisque le service de tests pousse plus loin ses investigations que les autres utilisateurs. Comme il est peu probable que vous receviez ce rapport de bogue précis de la part d'une autre source, vous avez vraiment intérêt à le préserver et à le rendre disponible au projet public.

Les membres du service qualité peuvent directement remplir un rapport dans le suivi public de problèmes, s'ils sont suffisamment à l'aise avec. Sinon un intermédiaire (en général l'un des développeurs) peut « traduire » le rapport interne du service de tests et le répertorier dans le suivi public. Traduire signifie simplement « décrire le bogue en ne faisant pas référence à des informations confidentielles du client » (la manière de le faire survenir peut utiliser des données du client, avec son accord évidemment).

Il est préférable, d'une certaine manière, que le service qualité remplisse directement les rapports dans le système de suivi public. L'implication de votre entreprise dans le projet gagne en visibilité : des rapports de bogue utiles consolideront sa crédibilité autant que n'importe quelle contribution technique. Un lien direct entre les développeurs et l'équipe de tests s'établit. Par exemple, si l'équipe qualité interne surveille le système de suivi public, un développeur peut valider un correctif pour un bogue d'extensibilité (pour le test duquel le développeur peut manquer des ressources nécessaires), en ajoutant une note au rapport demandant à l'équipe qualité de vérifier si le correctif a l'effet désiré. Quelques développeurs opposeront une certaine résistance, les programmeurs ont tendance à voir le service qualité,

au mieux, comme un mal nécessaire. L'équipe qualité peut facilement surmonter cette réticence en trouvant des bogues importants et en remplissant des rapports compréhensibles. L' interaction directe entre l'équipe qualité et l'équipe de développement perd nettement de son intérêt si leurs rapports ne sont pas au moins aussi bons que ceux provenant de la communauté d'utilisateurs réguliers.

Quoi qu'il en soit, une fois qu'un rapport public existe, le rapport interne originel devrait simplement y faire référence pour ce qui est du contenu technique. La direction et les développeurs payés peuvent continuer à annoter le rapport interne avec des commentaires particuliers à l'entreprise comme ils le jugent nécessaire, mais ils doivent continuer à utiliser le rapport public pour les informations qui devraient être disponibles à tous.

En vous engageant dans cette voie vous devez vous attendre à plus de contraintes de gestion. Maintenir deux rapports pour un bogue représente, naturellement, plus de travail que de maintenir un seul rapport. En contrepartie, beaucoup plus de codeurs verront ce rapport et pourront apporter leur contribution à la solution.

Conseils juridiques et protection

Les organisations, qu'elles soient ou non à but lucratif, sont presque les seules entités à prêter attention aux problèmes juridiques dans les logiciels libres. Les développeurs pris individuellement comprennent souvent les nuances entre les différentes licences Open Source, mais ils n'ont, en général, ni le temps ni les ressources pour maîtriser le détail des lois sur le droit d'auteur, les marques déposées et les brevets. Si votre entreprise comprend un service juridique, il peut contrôler les droits d'auteur du code et aider les développeurs à comprendre les problèmes possibles liés aux brevets et aux marques déposées. Dans le chapitre 9, nous aborderons plus en détail la forme exacte que peut prendre cette aide. Assurez-vous surtout que la communication entre le service juridique et la communauté de développement, si elle existe, se fait en tenant compte des différences entre les deux parties. Il arrive que ces deux groupes s'ignorent, chacun pensant que l'autre n'a pas les connaissances spécifiques requises. Un intermédiaire (en général un développeur ou alors un avocat ayant des connaissances techniques) pourra faire le lien entre les deux aussi longtemps que nécessaire.

Documentation et convivialité

La documentation et la convivialité sont deux points faibles connus des projets Open Source, bien qu'à mon avis, au moins en ce qui concerne la documentation, la différence entre les logiciels libres et propriétaires soit souvent exagérée. Néanmoins, par expérience, on constate que beaucoup de projets Open Source n'ont pas de documentation extraordinaire et ne recherchent pas la convivialité.

Si votre entreprise veut aider à combler ces lacunes dans un projet, elle devrait chercher à engager des contributeurs qui ne sont pas des développeurs réguliers du projet, mais qui seront capables d'interagir de façon productive avec les développeurs. Il vaut mieux ne pas engager de développeurs actifs pour deux raisons : d'abord, vous ne rognerez pas sur le temps de développement, et ensuite, ceux qui sont au plus près du logiciel, ne sont, en général, pas les mieux placés pour écrire la documentation ou pour en accroître la convivialité

parce qu'ils ont du mal à voir le logiciel du point de vue d'une personne extérieure au développement.

Dans tous les cas, les personnes travaillant sur ces problèmes seront amenées à communiquer avec les développeurs. Trouvez des personnes qui ont un niveau technique suffisant pour parler à l'équipe de programmation, mais qui ne sont pas autant impliquées dans le logiciel afin de pouvoir s'identifier aux utilisateurs « normaux ».

Un utilisateur de niveau moyen est certainement la personne la plus à même d'écrire une bonne documentation. En fait, après la publication de la première édition de ce livre, j'ai reçu cet e-mail d'un développeur Open Source qui s'appelle Dirk Reiners :

> Un commentaire sur :L'argent : :Documentation et convivialité : quand nous avons eu de l'argent à dépenser, et que nous avons décidé qu'un tutoriel pour débutant était ce dont nous avions besoin en priorité, nous avons engagé un utilisateur de niveau moyen pour l'écrire. Il était passé par la phase d'apprentissage suffisamment récemment pour se souvenir des problèmes, mais il les avait surmontés, il savait donc comment les décrire. Cela lui a permis d'écrire quelque chose, et les développeurs n'avaient plus que des petites corrections à apporter concernant ce qu'il n'avait pas écrit correctement, mais il couvrait quand même ces choses "évidentes" que les développeurs auraient omises. C'était vraiment la personne idéale pour nous, car son boulot était de présenter le logiciel à d'autres personnes (des étudiants), il rassemblait donc l'expérience de beaucoup d'utilisateurs, ce qui était une coïncidence heureuse pour nous mais certainement assez rare aussi.

Fournir hébergement / bande passante

Pour un projet qui n'utilise pas les services d'une forge (voir la section appelée « Forges » dans le chapitre 3), fournir un serveur, une connexion réseau et, mieux encore, une aide à l'administration système, peut être d'une importance primordiale. Même si c'est tout ce que votre organisation peut faire pour le projet, cela peut être une manière relativement efficace d'entretenir une bonne image, mais vous n'obtiendrez en retour aucune influence sur les décisions prises par le projet.

Vous pouvez sûrement espérer une bannière de publicité, ou un signe de reconnaissance sur la page principale du projet, remerciant votre entreprise pour la mise à disposition de l'hébergement. Si vous faites en sorte que l'adresse Web du projet soit sous le nom de votre entreprise, vous obtiendrez une association un peu plus marquée simplement au travers de l'URL. La plupart des utilisateurs penseront que le logiciel a un rapport avec votre entreprise, même si vous n'apportez rien du tout au développement. Mais les développeurs ne sont pas dupes, et ne seront pas forcément satisfaits que vous hébergiez le projet, sauf si vous

apportez d'autres ressources que simplement de la bande passante. Après tout, les possibilités d'hébergement sont nombreuses, maintenant. La communauté pourra au final décider que ce mérite indu ne vaut pas la commodité apportée par l'hébergement offert, et décidera de déménager le projet ailleurs. Donc si vous voulez offrir un hébergement, faites-le, mais prévoyez de vous impliquer encore plus prochainement, ou soyez circonspect à propos du niveau d'engagement que vous prétendez fournir.

8. Marketing

N'en déplaise aux développeurs Open Source, le marketing fonctionne. Une bonne campagne marketing peut alimenter le bouche à oreille autour d'un produit Open Source, au point que même les codeurs les plus bornés se mettent à voir d'un bon œil le logiciel pour des raisons qu'ils ne peuvent pas vraiment définir. Une description détaillée de la course à l'armement du marketing en général n'est pas le sujet de ce livre. Toute entreprise impliquée dans un logiciel libre réfléchira, tôt ou tard, aux moyens de se vendre, de vendre le logiciel ou de monnayer son investissement dans celui-ci. Les conseils ci-dessous vous aideront à éviter les pièges classiques dans ce cas, voir aussi la section appelée « Publicité » dans le chapitre 6.

Souvenez-vous que vous êtes sous surveillance

Afin de conserver la communauté de développeurs bénévoles de votre côté, il est très important de ne rien dire qui ne soit vérifiable. Contrôlez scrupuleusement toutes vos affirmations et donnez au public les moyens de les vérifier de son côté. La vérification indépendante et rapide compose une grande partie du domaine de l'Open Source et cela s'applique au-delà du code.

Évidemment, personne ne conseillerait aux entreprises de donner des informations non-vérifiables. Mais dans les activités Open Source, le nombre de ceux qui ont les compétences pour vérifier ces affirmations est exceptionnellement élevé, des personnes qui ont probablement une connexion à Internet à haut débit et les contacts qu'il faut pour publier leurs découvertes et faire des dégâts si elles le veulent. Quand une grande industrie chimique pollue une rivière, c'est vérifiable, mais seulement par des scientifiques formés, qui peuvent être réfutés par les scientifiques de cette industrie chimique. Le public, perplexe se demandera qui croire. À l'opposé, votre comportement dans le monde Open Source n'est pas seulement visible et enregistré, il est aussi facile pour beaucoup de personnes de le vérifier indépendamment, d'aboutir à leurs propres conclusions et de propager ces conclusions par le bouche à oreille. Ces voies de communication sont déjà établies, c'est l'essence même du fonctionnement de l'Open Source, et elles peuvent être utilisées pour transmettre toutes sortes d'informations. Vous défendre est en général très difficile, voire impossible, en particulier quand ce que disent les autres est vrai.

Par exemple, vous pouvez dire que votre organisation a « fondé le projet X » si vous l'avez effectivement fait. Mais ne vous proclamez pas « créateurs de X » si la majeure partie du code a été écrite par des développeurs étrangers à votre projet. À l'inverse, n'affirmez pas

que vous avez une communauté de développeurs volontaires profondément engagée si, par une simple vérification des dépôts, n'importe qui peut constater que l'écrasante majorité des modifications sont apportées par vos employés, et non pas par des volontaires extérieurs à votre entreprise.

Il n'y a pas si longtemps, j'ai vu une annonce faite par une entreprise informatique très connue affirmant qu'elle allait publier un paquet logiciel important sous une licence Open Source. Quand la première annonce a été faite, j'ai jeté un œil au dépôt de gestion de versions, désormais public, et j'ai vu qu'il ne contenait que trois révisions. En d'autres termes, ils avaient seulement importé le code source mais presque rien n'avait changé depuis. Ce n'est pas inquiétant en soi, ils venaient juste de faire l'annonce après tout. Il ne fallait pas s'attendre à voir tout de suite une activité frénétique.

Un peu plus tard, ils ont fait une nouvelle annonce. Voilà ce qu'elle disait, les noms et les numéros de version ont été remplacés par des pseudonymes :

> *« Nous sommes heureux d'annoncer qu'après avoir subi des tests rigoureux réalisés par la Singer Community, Singer 5 pour Linux et Windows est maintenant prêt pour une utilisation professionnelle. »*

Curieux de savoir ce qu'ils avaient découvert par leurs « tests rigoureux », je suis repassé par le dépôt pour y voir l'historique des changements récents. Le projet en était toujours à la version 3. Apparemment, ils n'avaient pas trouvé un seul bogue qui valait la peine d'être corrigé avant la sortie ! En pensant que les résultats des tests de la communauté avaient dû être enregistrés quelque part, j'ai ensuite examiné le système de suivi de bogues. Il y avait exactement six problèmes à traiter, dont quatre l'étaient déjà depuis quelques mois.

Cela défie l'entendement. Quand les testeurs s'attaquent à de gros morceaux compliqués d'un logiciel pendant un certain temps, ils trouvent forcément des bogues. Même si les correctifs pour ces bogues ne sont pas inclus dans la version à venir, on s'attend tout de même à voir une certaine activité dans le logiciel de gestion de versions, activité résultant du processus de test, ou au moins de nouveaux rapports. Pourtant, selon toute vraisemblance, rien ne s'était passé entre l'annonce du passage à une licence Open Source et la première version Open Source.

Le problème n'est pas que l'entreprise ait menti à propos des tests faits par la communauté. Je ne sais pas du tout s'ils mentaient ou non. Mais ils ne se rendaient pas compte qu'un observateur pouvait voir ça comme un gros mensonge. Puisque ni le dépôt de gestion de versions ni le suivi de problèmes ne donnent d'informations indiquant que le prétendu test rigoureux a eu lieu, l'entreprise n'aurait pas dû faire l'annonce du tout, ou alors, en fournissant un lien clair vers des résultats tangibles des tests (« Nous avons trouvé 278 bogues, cliquez ici pour plus de détails »). Ce dernier aurait permis à n'importe qui de mesurer l'activité de la communauté très rapidement. En l'état, cela m'a pris plusieurs minutes pour déterminer que, quoi qu'aient pu être ces tests, ils n'avaient pas laissé de traces aux endroits habituels. Ce n'est pas beaucoup demander, et je suis sûr de ne pas être le seul à avoir pris la peine de faire cette recherche.

La transparence et la « vérifiabilité » sont aussi nécessaires pour formuler des remerciements précis. Voir la section « Remerciements » dans le chapitre 8 pour en savoir plus à ce propos.

N'attaquez pas les produits Open Source concurrents

Abstenez-vous de critiquer les logiciels Open Source concurrents. Vous pouvez tout à fait exposer des faits négatifs, c'est-à-dire des affirmations facilement vérifiables, comme c'est souvent le cas dans les bons tableaux comparatifs. Mais il y a deux raisons pour lesquelles il vaut mieux éviter toute légèreté lorsqu'on diffuse des critiques négatives. Premièrement, elles sont susceptibles d'initier une guerre malsaine qui éloigne des discussions productives. Et surtout, deuxièmement, il se peut que certains développeurs volontaires de votre projet travaillent également sur le projet concurrent. C'est plus probable qu'il n'y paraît de prime abord : les projets appartiennent au même domaine (c'est pourquoi ils sont en compétition) et les développeurs ayant des compétences dans ce domaine peuvent apporter leur contribution partout où leur expérience est utile. Même quand il n'y a pas de chevauchement direct, il est probable que les développeurs sur votre projet sont au moins en relation avec les développeurs de l'autre projet. Des messages commerciaux trop négatifs pourraient affecter les liens personnels constructifs qu'ils entretiennent.

S'attaquer aux produits *closed source* concurrents semble être plus largement accepté dans le monde Open Source, en particulier quand ce sont des produits Microsoft. Personnellement, je déplore cette tendance (bien qu'encore une fois, il n'y ait aucun mal à faire une comparaison directe basée sur des faits), non seulement parce que c'est grossier, mais aussi parce qu'il peut être dangereux pour un projet de commencer à croire en sa propre popularité, et, de ce fait, ignorer que les rivaux peuvent en fait être meilleurs. De manière générale, faites attention aux effets que peuvent avoir les annonces commerciales sur votre propre communauté de développement. Le public peut devenir tellement enthousiaste à cause du soutien des dollars du marketing, qu'il ne voit plus objectivement les forces et les faiblesses du logiciel. Il est normal, et même attendu, de la part des développeurs de l'entreprise de montrer un certain détachement par rapport aux annonces commerciales, même dans les forums publics. Évidemment, ils ne devraient pas contredire le message commercial directement (à moins qu'il ne soit en fait faux, mais il est souhaitable que ce genre de faux-pas soit détecté plus tôt). Mais ils peuvent le tourner en dérision de temps en temps, c'est un moyen de ramener le reste de la communauté sur Terre.

Communication

La capacité à écrire avec clarté est peut-être la qualité la plus importante que l'on puisse avoir dans un environnement Open Source. Sur le long terme, elle compte davantage que les compétences en programmation. S'il a des difficultés à communiquer, un programmeur aura beau être génial, il ne pourra s'occuper que d'une chose à la fois, et il n'est même pas certain d'obtenir l'attention désirée. En revanche, un piètre programmeur doué en communication peut coordonner plusieurs personnes et les convaincre de faire de multiples choses, tout en ayant un impact significatif sur la direction et la dynamique du projet.

Savoir bien communiquer et produire du bon code sont deux choses différentes. Un bon programmeur saura bien décrire les questions techniques, mais ce n'est là qu'une infime partie du travail de communication que requiert le projet. Savoir s'identifier à son auditoire, voir ses propres messages et commentaires comme les autres les voient, et les amener à percevoir les leurs avec la même objectivité est bien plus important. Se rendre compte qu'un processus de communication ne fonctionne plus correctement parce qu'il n'est plus adapté au nombre grandissant d'utilisateurs, par exemple, et prendre le temps d'y remédier est tout aussi important.

Tout ceci est évident d'un point de vue théorique ; ce qui complique les choses en pratique, c'est que les environnements de développement de logiciel libre offrent une variété déconcertante de publics et de mécanismes de communication. Faut-il exprimer telle remarque par un message dans la liste de discussion, par une note dans le traqueur de bogues ou par un commentaire dans le code ? Pour répondre à une question dans un forum public, quel degré de connaissance faut-il supposer chez« le lecteur » étant donné que ce dernier n'est pas seulement celui qui a posé la question au départ mais aussi tous ceux qui peuvent lire la réponse ? Comment les développeurs peuvent-ils rester en contact de manière constructive avec les utilisateurs sans être submergés de demandes sur les fonctionnalités, de rapports

de bogue infondés et de discussions généralistes ? Comment déterminer le moment où un support a atteint la limite de ses capacités et que faire pour y remédier ?

Les solutions à ces problèmes sont souvent ponctuelles, car toute solution particulière devient, à l'occasion, obsolète quand le projet grandit ou que sa structure évolue. Elles sont souvent ad hoc, car ce sont des réponses improvisées à des situations dynamiques. Chaque participant est responsable de la bonne conduite du projet, il faut savoir analyser les problèmes et trouver des solutions. L'aide aux autres dans ce domaine constitue une part importante de la gestion d'un projet Open Source. Dans les sections suivantes, nous verrons comment gérer votre propre communication, et comment faire du maintien des mécanismes de communication une priorité pour chacun des membres du projet [1].

1. Vous êtes ce que vous écrivez

C'est un fait : sur Internet on ne vous connaît qu'au travers de vos écrits ou de ce que l'on écrit sur vous. Vous pouvez être brillant, subtil et charismatique mais si vos e-mails sont décousus et mal structurés, on supposera que vous l'êtes vous aussi ! Ou alors vous êtes réellement incohérent et désorganisé, mais personne n'a besoin de le savoir, du moment que vos messages sont clairs et informatifs.

Rédiger vos messages avec soin sera toujours payant. Jim Blandy, hacker de logiciel libre de longue date, raconte l'histoire suivante :

> *« En 1993, je travaillais pour la Free Software Foundation, et nous étions en train de faire des bêta tests pour la version 19 de GNU Emacs. Nous sortions une version bêta par semaine, à peu près, pour que des utilisateurs l'essaient et nous envoient des rapports de bogue. Il y avait ce gars qu'aucun de nous n'avait rencontré personnellement, mais qui avait fait du gros boulot : ses rapports de bogue étaient toujours clairs et nous menaient droit au problème, de même, quand il proposait une correction, c'était presque toujours exact. Il était vraiment bon.*
>
> *Or, avant d'utiliser le code écrit par une personne extérieure, la FSF fait signer un papier stipulant que l'auteur cède ses droits à la FSF sur ce bout de code. Se contenter d'injecter le code de parfaits inconnus mène de manière certaine à un désastre juridique.*
>
> *Donc, j'envoyais les formulaires à ce type en disant : 'Voici quelques papiers dont nous avons besoin, voici ce qu'ils veulent dire, signez celui-ci, faites signer cet autre à votre employeur, après quoi nous pourrons intégrer vos corrections. Merci beaucoup.'*
>
> *Il me renvoya un message qui disait : 'Je n'ai pas d'employeur.'*

1. Quelques recherches universitaires intéressantes sont parues sur le sujet, je vous renvoie par exemple à *Group Awareness in Distributed Software Development* par Gutwin, Penner et Schneider (cet ouvrage était disponible en ligne mais il semble qu'il ne le soit plus, temporairement en tout cas ; utilisez un moteur de recherche pour le trouver).

Je répondis : 'Pas de problème, faites-le signer par votre université et renvoyez-le nous.'

Après un certain temps, il m'écrivit ceci : 'En fait… j'ai treize ans et j'habite chez mes parents.' »

Personne ne se doutait de l'âge réel de ce jeune homme, parce qu'il n'écrivait pas comme un gamin de treize ans. Vous trouverez dans ce qui suit quelques idées pour que vos écrits, eux aussi, fassent bonne impression.

Structure et format

Ne tombez pas dans la facilité en utilisant le langage SMS. Faites des phrases complètes, utilisez des majuscules en début de phrase, et aérez le texte en revenant à la ligne pour les nouveaux paragraphes. C'est particulièrement important pour les e-mails, ou tout autre texte long que vous aurez à écrire. Sur IRC, ou d'autres forums éphémères, on peut se passer des majuscules, on peut utiliser des abréviations pour les expressions courantes, etc. Évitez simplement de garder ces habitudes sur les forums permanents plus formels. Les e-mails, la documentation, les rapports de bogue, ou tout autre texte n'ayant pas une durée de vie limitée, doivent être rédigés en respectant les règles d'orthographe et de grammaire et avoir une structure narrative cohérente. Ce n'est pas pour le plaisir de suivre des règles arbitraires, car justement, elles ne le sont pas : elles ont évolué jusqu'à leur forme présente afin de rendre le texte plus lisible, et vous devriez vous y tenir précisément pour cette raison. La lisibilité est importante. Non seulement davantage de personnes vous comprendront, mais vous montrerez de surcroît que vous êtes de ceux qui prennent le temps de communiquer clairement : quelqu'un digne d'intérêt.

Pour les e-mails en particulier, des développeurs Open Source expérimentés ont mis au point certaines conventions :

Envoyez vos e-mails au format texte uniquement, pas d'HTML, pas de texte enrichi ou d'autres formats que pourraient ne pas prendre en charge certains logiciels de messagerie. Formatez les lignes pour qu'elles fassent environ 72 colonnes de large. Ne dépassez pas 80 colonnes, taille qui s'est imposée comme la largeur standard des terminaux (certaines personnes peuvent utiliser des terminaux plus larges, mais personne n'utilise de terminal moins large). En écrivant vos e-mails en moins de 80 colonnes, vous laissez de l'espace pour plusieurs niveaux de citation qui seront ajoutés avec les réponses des autres participants, sans perdre la mise en forme de votre texte.

Utilisez de vrais retours à la ligne. Certaines messageries utilisent une sorte de fausse mise en page qui fait que, quand vous composez votre e-mail, les retours à la ligne s'affichent mais sans être vraiment là. Quand vous envoyez votre e-mail, ces retours à la ligne, que vous pensiez avoir insérés, peuvent manquer et la mise en page sera étrange chez certaines personnes. Si votre messagerie utilise de faux retours à la ligne, regardez s'il n'y a pas une option à cocher pour obtenir les vrais retours à la ligne lors de l'écriture.

Lorsque vous ajoutez les résultats d'une commande, des fragments de code ou tout autre texte préformaté, marquez un décalage prononcé pour que même les plus myopes puissent distinguer la frontière entre votre prose et le texte cité (je ne prévoyais pas d'ajouter ce

conseil lorsque j'ai commencé ce livre, mais j'ai récemment observé, sur plusieurs listes de diffusion, un mélange de textes provenant de différentes sources sans marquer de distinction claire entre le texte et le code. Le résultat est très frustrant. Les messages sont moins clairs, et ils ne véhiculent pas une bonne image de leurs auteurs).

Lorsque vous citez le message de quelqu'un d'autre, insérez vos réponses là où elles sont attendues, en plusieurs endroits si besoin, et retirez les parties du message original qui vous sont inutiles. Vous pouvez répondre directement (c'est-à-dire donner votre réponse directement au-dessus du texte cité), si vous rédigez un commentaire rapide qui s'applique au message entier. Autrement, vous devriez citer les parties pertinentes du texte original suivies de vos réponses.

Réfléchissez bien à la ligne *Sujet :* lorsque vous écrivez un nouvel e-mail. C'est la ligne la plus importante de votre courrier car elle permet à chaque participant du projet de décider s'il doit le lire ou non. Les logiciels de messagerie modernes organisent les messages liés en un fil de discussion qui peut être défini, non seulement par un sujet commun, mais aussi par d'autres en-têtes (qui ne sont pas toujours visibles). Par conséquent, si un fil commence à dévier vers une autre discussion, vous pouvez — et devez — modifier le sujet de façon adéquate au moment de répondre. L'intégrité du fil de discussion sera préservée grâce aux autres en-têtes, mais le nouveau sujet permettra aux personnes suivant le fil de loin de savoir que le sujet a dérivé. De même, si vous voulez vraiment aborder un nouveau sujet, commencez par un nouveau courrier, ne vous contentez pas de répondre à un e-mail précédent en modifiant simplement le sujet. Si vous faites cela, votre e-mail sera toujours groupé dans le même fil que celui auquel vous répondiez et induira les autres en erreur. Encore une fois, ce n'est pas qu'un problème de perte de temps, votre crédibilité de personne maîtrisant les outils de communication s'en trouvera affectée.

Contenu

Des e-mails bien écrits attirent les lecteurs, mais c'est le contenu qui retient leur attention. Il n'y a évidemment aucune formule magique à ce niveau-là, alors respecter quelques principes sera déjà un bon début.

Dans tous les projets Open Source actifs, des tonnes d'informations sont générées, auxquelles, autant que faire se peut, vous devez fournir un accès direct. Cela vaut la peine de prendre deux minutes de plus pour aller chercher l'URL d'un fil de discussion particulier dans les archives des listes de diffusion afin d'éviter aux lecteurs d'avoir à le faire. S'il vous faut 5 à 10 minutes pour résumer les conclusions d'une discussion complexe et replacer votre message dans son contexte pour qu'il soit mieux compris, alors faites-le. Voyez les choses de la manière suivante : plus un projet a de succès, plus le ratio lecteur / rédacteur augmente. Si chacun de vos messages est vu par n personnes, plus n augmente, plus vos quelques efforts supplémentaires seront rentables. De plus, en voyant que vous vous imposez cette rigueur, les autres essaieront d'en faire autant dans leurs propres messages. Le résultat idéal est l'amélioration de l'efficacité générale : le projet préfèrera toujours qu'une personne fasse l'effort pour tous que l'inverse.

Ne vous lancez pas dans des hyperboles. L'exagération dans les messages en ligne est une course à l'armement classique. Par exemple, une personne rapportant un bogue pourrait craindre que les développeurs n'y prêtent pas suffisamment attention, il le décrira alors

comme un problème sévère qui l'empêche (lui et tous ses amis, collègues, cousins) d'utiliser le logiciel de manière productive, alors que le désagrément, et en fait, bénin. Cette dramatisation ne relève pas uniquement des utilisateurs, les programmeurs font souvent la même chose au cours de débats techniques, en particulier quand le désaccord repose plus sur une question de goût que d'exactitude :

> *« Faire ainsi rendrait le code totalement illisible. La maintenance deviendrait un cauchemar comparé à la proposition de A. Nonyme… »*

Le même argument devient plus convaincant lorsqu'il est formulé sur un ton plus neutre :

> *« Ça fonctionne, mais c'est loin d'être la panacée en terme de lisibilité et de dépannage, je pense. La proposition de A. Nonyme nous épargne ces problèmes car… »*

Vous ne parviendrez pas à éradiquer complètement les figures de rhétorique, ce qui n'est, d'ailleurs, généralement pas nécessaire. Comparée à d'autres formes de mauvaise communication, l'hyperbole n'est pas si gênante, elle retombe en général sur son auteur. Les destinataires peuvent compenser, mais l'expéditeur perd un peu plus de crédibilité, à chaque fois. Alors, pour le bien de votre propre influence au sein du projet, faites preuve de modération. Ainsi, quand vous voudrez vraiment être pris au sérieux sur un problème grave, vous serez écouté.

Relisez deux fois, avant de l'envoyer, tout message plus long qu'un paragraphe de taille moyenne même après vous être déjà dit que c'était bon la première fois. Ce conseil doit sembler familier à quiconque a pris des cours de composition, mais c'est particulièrement important pour les discussions en ligne. Parce que l'écriture d'un e-mail peut être très discontinue (pendant l'écriture d'un message vous aurez peut-être à lire d'autres courriers, à parcourir certaines pages Web, à lancer une commande pour voir le message de débogage, etc.), il est très facile de perdre le fil de ce que l'on écrivait. Un message composé dans ces conditions et non relu avant d'être envoyé se repère facilement, au grand dam de son auteur (ou du moins peut-on l'espérer). Prenez le temps de vous relire. Plus vos messages seront cohérents, plus ils seront lus.

Ton

Après avoir écrit des milliers de messages, vous observerez sûrement une simplification de votre style. Cela semble être la norme dans la plupart des forums techniques, et intrinsèquement, il n'y a rien de mal à cela. Un degré de simplification inacceptable pour des interactions sociales habituelles est simplement la norme des hackers de logiciels libres. Voilà une réponse, tirée d'une liste de diffusion, à propos d'un logiciel libre de gestion de contenu, je cite la réponse complète :

> Est-ce que tu pourrais développer un peu plus les problèmes que tu as rencontrés, etc. ?
> Et aussi :

> Quelle version de Slash utilises-tu ? Je n'ai pas réussi à le savoir avec ton premier message.
>
> Comment as-tu construit exactement la source de apache/-mod_perl ? As-tu essayé le correctif Apache 2.0 qui a été posté sur slashcode.com ?
>
> Shane

Alors *ça*, c'est succinct ! Pas de bonjour, pas d'au revoir autre que sa signature, le message en lui-même n'est qu'une série de questions aussi brèves que possible. Sa seule phrase déclarative n'est qu'une critique implicite de mon message. Et pourtant j'étais content de voir la réponse de Shane, car j'ai vu dans sa brièveté la marque d'une personne très occupée. Le simple fait qu'il me pose des questions plutôt que d'ignorer mon message signifiait qu'il voulait bien accorder un peu de temps à mon problème.

Par contre, tous les lecteurs ne réagiront pas forcément bien face à ce style, tout dépend de la personne et du contexte. Par exemple, si quelqu'un vient juste d'envoyer un message pour reconnaître son erreur (supposons qu'il a écrit un bogue), et que vous savez, par des expériences passées, que cette personne manque de confiance en elle, vous devriez inclure dans votre réponse que vous comprenez ce qu'elle ressent. L'analyse de la situation par votre œil expert vous conduit à une réponse concise et aussi succincte que possible. Mais à la fin, concluez par quelque chose indiquant que votre brièveté n'est pas à prendre pour de la froideur. Par exemple, si vous venez juste de donner des tas de conseils pour aider la personne à réparer le bogue, terminez par « Bonne chance <votre nom> » pour indiquer que vous êtes de tout cœur avec elle et que vous n'êtes pas fâché. Un petit smiley bien placé ou toute autre émoticône suffit souvent à rassurer un interlocuteur, pensez-y.

Il peut paraître étrange de se concentrer autant sur le bien-être des participants que sur le contenu de leur message, mais pour dire les choses comme elles sont : le bien-être affecte la productivité. Les sentiments sont importants pour d'autres raisons encore, ainsi, même d'un point de vue strictement pratique, on constate que ceux qui sont mécontents écrivent de moins bons logiciels et avec un rendement plus faible. Mais de par leur nature même, les communications électroniques dévoilent peu les vrais sentiments d'une personne. Il faudra vous appuyer sur vos déductions en vous basant sur a) comment une personne lambda se sentirait dans cette situation et b) ce que vous savez de cette personne par vos interactions passées. Certains préfèrent adopter une attitude plus détachée, et s'appuient sur ce qui est dit ouvertement. L'idée étant que si un participant ne dit pas d'emblée que quelque chose le dérange, alors personne ne devrait faire comme si c'était le cas. Je n'adhère pas à cette approche pour plusieurs raisons. D'une part, les utilisateurs ne se comportent pas ainsi dans la vie, pourquoi devraient-ils le faire en ligne ? Et d'autre part, puisque les interactions se font sur des forums publics, la tendance à masquer les émotions est plus forte encore. Plus précisément, on exprime facilement ses émotions envers les autres, comme la gratitude ou l'énervement, mais pas les sentiments intérieurs comme l'insécurité ou la fierté. Et pourtant, la plupart des participants travaillent mieux en sachant les autres au courant de leur état d'esprit. En faisant attention aux petits détails, vous vous tromperez rarement, et vous arriverez à motiver les contributeurs pour qu'ils restent plus impliqués qu'ils ne pourraient l'être autrement.

Je ne veux pas dire par là que votre rôle soit celui d'un thérapeute de groupe toujours prêt à aider les membres à être en harmonie avec leurs émotions. Mais en faisant attention aux comportements sur le long terme, vous commencerez à bien les connaître même si vous ne les avez jamais vraiment rencontrés. Et, en contrôlant votre propre ton, vous pourrez obtenir une influence surprenante sur le moral des participants, au profit du projet.

Reconnaître la vulgarité

L'une des caractéristiques de la culture Open Source est sa notion particulière de ce qui est ou n'est pas vulgaire. Ce qui suit n'est pas propre au développement de logiciels libres, pas même aux logiciels en général, mais relève des conventions familières à n'importe qui travaillant dans les mathématiques, les sciences dures ou les disciplines d'ingénierie. Le logiciel libre, avec ses frontières perméables et le flux constant de nouveaux venus, est un environnement où ces conventions s'appliqueront également à ceux qui n'y sont pas accoutumés.

Commençons par voir ce qui n'est *pas* offensant :

Les critiques techniques, même adressées directement et sans prendre de gants, ne sont pas offensantes. Cela peut même être une sorte de flatterie. En effet, le jugement implique que la personne visée mérite qu'on la prenne au sérieux. Alors qu'il aurait été plus facile de simplement ignorer le message, prendre le temps de l'examiner devient un compliment (à moins que la critique ne se résume qu'à une attaque *ad hominem* ou toute autre forme de vulgarité évidente bien sûr).

Les questions brutes posées directement, comme les questions posées par Shane dans l'exemple précédent, ne sont pas offensantes non plus. Des questions, qui dans un autre contexte pourraient sembler froides, rhétoriques ou même railleuses, sont souvent posées sérieusement, et n'ont pas d'autre but que d'obtenir les informations aussi rapidement que possible. La célèbre question du support technique « Votre ordinateur est-il bien branché ? » en est un exemple classique. La personne du support technique a vraiment besoin de savoir si votre ordinateur est bien branché, et après quelques jours à faire ce travail, elle s'exaspère d'introduire sa question par de polies courbettes (« Je vous prie de m'excuser, je voudrais juste vous poser plusieurs questions pour éliminer quelques possibilités. Certaines vous paraîtront très simples, mais s'il vous plaît, faisons-le ensemble… »). Arrivée à un certain point, elle laissera tomber la forme pour arriver directement au but : est-il branché ou non ? Des questions équivalentes sont posées sans cesse sur les listes de diffusion des logiciels libres. Le but n'est pas de blesser le destinataire, mais d'éliminer rapidement les explications les plus évidentes (et peut-être les plus courantes). Comprendre ceci et répondre en conséquence marque des points et démontre une certaine ouverture d'esprit. Mais ceux qui ne réagissent pas bien ne devraient pas en être réprimandés pour autant. Il s'agit d'un choc culturel, ce n'est pas la faute de quelqu'un en particulier. Expliquez amicalement que vos questions (ou critiques) n'ont pas de sens caché, elles sont simplement posées pour obtenir (ou transmettre) des informations aussi efficacement que possible, rien de plus.

Mais alors, qu'est-ce qui est vulgaire ?

Si on suit le principe selon lequel une critique technique est une forme de flatterie, ne pas apporter de jugement constructif peut être une forme d'insulte. Je ne parle pas simplement

d'ignorer le travail de quelqu'un, que ce soit une proposition, une modification ou le rapport d'un nouveau problème, etc. À moins d'avoir promis explicitement une réaction détaillée, ce n'est pas grave si vous ne réagissez pas du tout. Les gens penseront simplement que vous n'avez pas eu le temps de dire quoi que ce soit. Mais si vous réagissez, par contre, ne lésinez pas : prenez le temps de vraiment analyser les choses, fournissez des exemples concrets quand il le faut, creusez un peu les archives pour trouver d'anciens messages ayant un lien, etc. Et, si vous n'avez pas le temps de consentir à tous ces efforts, mais que vous devez tout de même écrire une réponse brève, dites-le ouvertement (« Je pense que le problème a déjà été rapporté, mais malheureusement je n'ai pas le temps de faire la recherche, désolé. »). Il est primordial de reconnaître l'existence de la norme culturelle, soit en la respectant, soit en reconnaissant ouvertement que quelqu'un ne s'y est pas tenu. Dans les deux cas la norme est renforcée. Mais ne pas respecter la norme sans expliquer pourquoi, revient à dire que le sujet (et ceux qui y participent) n'est pas digne de votre temps. Il vaut mieux montrer que votre temps est précieux en étant bref plutôt que paresseux.

Il y a bien d'autres formes de vulgarité évidemment, mais la plupart ne sont pas particulières au développement de logiciels libres : un peu de bon sens devrait suffire à les éviter. Reportez-vous également à la partie intitulée « Tuez la vulgarité dans l'œuf » dans le chapitre 2, si vous ne l'avez pas déjà fait.

Visage

Une région entière du cerveau est dédiée à la reconnaissance des visages. Elle est officieusement connue sous le nom de « zone fusocellulaire des visages » et ses aptitudes sont principalement innées, pas acquises. Il s'avère que la reconnaissance des individus est devenue si cruciale que nous avons développé du matériel spécialisé pour cela.

La collaboration en ligne est par conséquent une expérience psychologiquement étrange, car elle demande une coopération étroite entre des êtres humains qui n'ont quasiment jamais l'opportunité d'identifier les autres par les méthodes les plus intuitives et naturelles : la reconnaissance faciale en premier lieu, mais aussi le son de la voix, la posture, etc. Pour compenser ceci, essayez d'utiliser toujours le même pseudonyme. Il devrait être la première partie de votre adresse mail (la partie avant le signe @), votre nom d'utilisateur IRC, votre nom de *committer*, votre nom pour le suivi de problèmes et ainsi de suite. Ce nom est votre « visage » virtuel : un moyen d'identification court qui remplace au moins en partie votre visage, même si, malheureusement, il ne fait pas appel aux mêmes parties du cerveau.

Le pseudonyme devrait dériver logiquement de votre vrai nom (le mien par exemple est « kfogel »). Dans certains cas il sera accompagné par votre nom complet de toute façon, par exemple dans les en-têtes des mails :

> De : "Karl Fogel" <kfogel@undomaine.com>

En fait, il y a deux choses importantes ici. Comme nous l'avons dit précédemment, le pseudonyme rappelle votre vrai nom de manière intuitive, mais il faut aussi que votre vrai nom soit un *vrai* nom. Ce n'est pas un nom inventé comme :

> De : "Wonder Hacker" <wonderhacker@undomaine.com>

Un dessin très connu de Paul Steiner, paru dans le numéro du 5 juillet 1993 de *The New Yorker*, montre un chien devant un ordinateur qui regarde un autre chien et lui dit d'un air conspirateur : « Sur Internet, personne ne sait que tu es un chien ». C'est probablement ce genre de pensées qui sont à l'origine de beaucoup de pseudonymes glorifiants ou pseudo-cool que certains se donnent, comme si se donner le nom de « Wonder Hacker » amenait vraiment les autres à penser que vous êtes un hacker merveilleux. Mais ça n'efface pas la réalité : même si personne ne sait que vous êtes un chien, vous êtes tout de même un chien. Une identité fantaisiste sur Internet n'impressionne pas les lecteurs. Au contraire, cela peut les amener à penser que vous vous souciez plus de l'image que du fond, ou encore, que vous êtes peu sûr de vous. Utilisez votre vrai nom pour toutes les interactions, ou, si pour certaines raisons vous voulez préserver votre anonymat, trouvez-vous un nom qui semble normal et utilisez-le en permanence.

En plus de toujours montrer le même « visage en ligne », il y a quelques trucs que vous pouvez faire pour le rendre plus plaisant. Si vous avez un titre officiel (par exemple « docteur », « professeur », « directeur ») n'en faites pas étalage, ne le mentionnez même pas, sauf si cela a une certaine importance dans la conversation. Le monde des hackers en général, et la culture des logiciels libres en particulier, a tendance à voir les titres comme une mise à l'écart et un signe de manque d'assurance. Vous pouvez mettre votre titre dans la signature automatique des mails que vous envoyez, mais ne vous en servez jamais pour justifier votre position dans une discussion, vous pouvez avoir la certitude que cela se retournera contre vous. Vous voulez que les autres vous respectent en tant que personne, pas seulement pour votre titre.

En parlant de signature automatique : il faut qu'elle soit brève et efficace ou mieux : inexistante. Évitez les avertissements légaux insérés à la fin de chaque mail, en particulier lorsqu'ils expriment des idées incompatibles avec votre participation dans un projet de logiciel libre. Par exemple, le classique du genre qui suit apparaît à la fin de chaque message qu'un participant envoie à la liste de diffusion publique sur laquelle je suis :

> IMPORTANT NOTICE If you have received this e-mail in error or wish to read our e-mail disclaimer statement and monitoring policy, please refer to the statement below or contact the sender. This communication is from Deloitte and Touche LLP. Deloitte and Touche LLP is a limited liability partnership registered in England and Wales with registered number OC303675. A list of members' names is available for inspection at Stonecutter Court, 1 Stonecutter Street, London EC4A 4TR, United Kingdom, the firm's principal place of business and registered office. Deloitte and Touche LLP is authorised and regulated by the Financial Services Authority. This communication and any attachments contain information which is confidential and may also be privileged. It is for the exclusive :use of the intended recipient(s). If you are not the intended recipient(s) please note that any form of disclosure, distribution copying or use of this communication or the information in it or in any attachments

> is strictly prohibited and may be unlawful. If you have received this communication in error, please return it with the title "received in error" to IT.SECURITY.UK@deloitte.co.uk then delete the e-mail and destroy any copies of it. E-mail communications cannot be guaranteed to be secure or error free, as information could be intercepted, corrupted, amended lost, destroyed, arrive late or incomplete, or contain viruses. We do not accept liability for any such matters or their consequences. Anyone who communicates with us by e-mail is taken to accept the risks in doing so. When addressed to our clients, any opinions or advice contained in this e-mail and any attachments are subject to the terms and conditions expressed in the governing Deloitte and Touche LLP client engagement letter. Opinions, conclusions and other information in this e-mail and any attachments which do not relate to the official business of the firm are neither given nor endorsed by it.

Pour une personne qui vient poser quelques questions de temps en temps, cet avertissement énorme semble un peu étrange sans causer de problème à long terme. En revanche, si cette personne participe activement au projet, ce pavé légal aura un effet plus insidieux. Il véhicule au moins deux signaux destructeurs : cette personne ne maîtrise pas complètement ses outils, elle est enfermée dans une messagerie d'entreprise qui colle un message gênant à la fin de chaque message, sans trouver le moyen de contourner cela, et elle n'a pas ou peu de soutien de son entreprise pour ses activités dans le logiciel libre. C'est vrai que l'entreprise ne l'empêche pas véritablement de participer aux listes de diffusion publiques, mais elle rend ses messages vraiment peu accueillants, comme si le risque de laisser échapper des informations confidentielles prenait le pas sur toutes les autres priorités.

Si vous travaillez pour une société imposant une telle signature jointe à tous les courriers sortants, vous devriez envisager l'utilisation d'un compte de messagerie gratuite comme par exemple gmail.google.com, www.hotmail.com ou www.yahoo.com et l'utiliser pour le projet.

2. Éviter les pièges courants

Ne pas poster sans raison

Quand vous participez à un projet en ligne vous serez tenté de répondre à tout, c'est l'un des pièges classiques. Ne le faites pas. Tout d'abord, il risque d'y avoir plus de fils de discussions que vous ne pouvez en suivre, en tout cas une fois que le projet se sera développé. Deuxièmement, même dans les fils que vous avez décidé de suivre, la majeure partie de ce qu'écrivent les gens n'appelle pas de réponse. Dans les forums de développement plus particulièrement, trois types de messages prédominent :

1. des messages proposant quelque chose de pertinent,

2. des messages pour exprimer son soutien ou son opposition à quelques propos,
3. des messages récapitulatifs.

Aucun de ceux-là n'appelle concrètement de réponse, notamment si vous êtes pratiquement certain, compte-tenu de la teneur du fil jusque-là, que quelqu'un d'autre va probablement dire ce que vous auriez dit (ne craignez pas que les autres s'abstiennent et temporisent également, soyez-en sûr, il y a presque toujours quelqu'un qui aura envie de se jeter dans la mêlée). Une réponse doit être motivée par un objectif précis. Demandez-vous d'abord : est-ce que je sais où je veux en venir ? Puis : est-ce que ceci a des chances de se réaliser sans que je m'exprime ?

Il y a deux bonnes raisons pour intervenir dans un fil de discussion : a) quand on voit une faille dans une proposition et qu'on suppose être le seul à la voir et b) quand on voit que les autres ont du mal à communiquer et que l'on pense pouvoir éclaircir la situation en postant un message. On peut aussi poster un message simplement pour remercier quelqu'un, ou, pour dire « Moi aussi ! », car le lecteur peut comprendre immédiatement que ce genre de messages n'entraîne aucune réponse ni aucune action, en conséquence de quoi l'effort mental requis par le message se termine proprement quand le lecteur en a atteint la dernière ligne. Mais, même dans ce cas là, réfléchissez-y à deux fois avant de dire quelque chose : il vaut mieux que l'on pense que vous pourriez poster plus, et non que vous pourriez poster moins (pour plus de réflexions sur le comportement à adopter sur une liste très active, voir la seconde partie de l'Annexe C).

Fils de discussion productifs contre fils improductifs

Dans une liste de discussion animée, il y a deux impératifs. Le premier, évidemment, consiste à distinguer ce qui mérite votre attention de ce que vous pouvez ignorer. L'autre consiste à agir de manière à ne pas créer de bruit : vous voulez non seulement que vos propres messages soient pertinents, mais aussi qu'ils incitent les autres à en poster d'aussi pertinents ou bien à s'abstenir.

Pour voir comment y parvenir, examinons le contexte. Quelles sont les marques distinctives d'un fil improductif ?

– Quand les arguments déjà avancés sont répétés, comme si leur auteur pensait que personne ne les avait entendus la première fois.
– Quand les niveaux d'hyperbole et d'engagement augmentent au fur et à mesure que l'enjeu diminue.
– Quand la majorité des commentaires émanent de ceux qui ne font rien ou presque, tandis que les plus actifs gardent le silence.
– Quand plusieurs idées sont en discussion sans qu'il y ait de propositions claires (bien sûr, toutes les idées intéressantes commencent par une vision imprécise : ce qui compte, c'est la direction qu'elles prennent à partir de là. Est-ce que le fil a tendance à rendre la discussion plus concrète ou est-ce qu'il s'emberlificote dans des perspectives secondaires, voire latérales et des disputes ontologiques ?).

Le fait qu'un fil ne soit pas productif à son début ne le condamne pas nécessairement. Il peut s'agir d'un sujet important, auquel cas le fait que le fil piétine est d'autant plus troublant.

Redonner de l'intérêt à un fil de discussion utile sans faire de rentre-dedans est tout un art. Reprocher aux gens leur perte de temps, ou leur demander de ne poster que des messages constructifs, ne mène pas à grand-chose. Vous pouvez, bien sûr, le penser intérieurement, mais l'exprimer ouvertement serait insultant. En revanche, vous devez suggérer des pistes pour progresser : donner une direction, une voie à suivre qui mène au résultat escompté, sans pour autant avoir l'air d'imposer une conduite. Voici par exemple ce qu'il ne faut pas faire :

> *« Cette discussion ne mène nulle part. Pouvez-vous laisser tomber ce sujet jusqu'à ce que quelqu'un ait fait un correctif pour mettre en œuvre une de ces propositions ? Inutile de répéter toujours les mêmes choses. Le code parle plus que des mots les gars. »*

Voici qui est mieux :

> *« Plusieurs propositions ont été esquissées dans ce fil, mais aucune n'est suffisamment précise pour qu'on puisse voter pour ou contre. De plus, nous n'avançons aucune idée nouvelle, mais répétons ce qui a déjà été dit. Donc la meilleure chose, à ce point, serait que les messages à venir contiennent soit une spécification complète pour le comportement proposé, soit un correctif. Nous aurons ainsi au moins une action précise à entreprendre (c-à-d. nous mettre d'accord sur les spécifications ou appliquer le correctif et le tester). »*

Comparez les deux approches. La deuxième ne trace pas une ligne de démarcation entre vous et les autres, elle n'accuse personne de mener la discussion dans une impasse. Elle parle de « nous », ce qui est important, que vous ayez ou non participé auparavant à la discussion, car cela rappelle à chacun que, même pour ceux qui sont restés silencieux jusqu'ici, l'issue de la discussion compte. Elle expose pourquoi le fil ne va nulle part, mais le fait sans dénigrer ou porter de jugement, elle ne fait que constater des faits de manière neutre. Et surtout, elle offre une ligne d'action positive, de sorte que les gens, au lieu d'avoir l'impression que la discussion a été stoppée (frein contre lequel ils ne peuvent que s'insurger), auront l'impression qu'on leur offre un moyen d'amener la discussion dans une autre direction plus constructive, ce qu'ils accepteront spontanément.

On ne voudra pas toujours porter une discussion à un plus haut degré de productivité, parfois il est souhaitable qu'elle cesse. Le but de votre message est alors d'aboutir à l'un ou à l'autre. Si vous pouvez voir, d'après ce qui s'est dit dans le fil, que personne n'est prêt à s'engager dans la voie que vous proposez, alors votre message clôt effectivement le fil sans en avoir l'air. Évidemment, il n'y a aucun moyen infaillible pour clore un fil, et même s'il y en avait un, vous n'y auriez pas recours. Mais demander aux participants de faire des progrès ostensibles, ou d'arrêter de poster, est tout à fait légitime si vous le faites avec diplomatie. Gardez-vous d'étouffer des discussions prématurément toutefois. Une certaine dose de causette spéculative peut être productive selon le sujet mais demander à ce qu'elle aboutisse trop rapidement fera tourner court le processus créatif, et vous passerez pour un impatient.

Ne vous attendez pas à faire cesser un fil instantanément. Il y aura certainement quelques messages de plus après le vôtre, soit parce que les messages se sont croisés, soit parce que les gens veulent avoir le dernier mot. Il n'y a aucune raison de s'inquiéter, et il n'est pas nécessaire de poster un nouveau message. Laissez le fil s'éteindre, ou non suivant le cas. Vous ne pouvez pas avoir le contrôle absolu : d'un autre côté, statistiquement, vous pouvez espérer que vos rappels portent leurs fruits après les avoir réitérés sur plusieurs fils.

Plus le sujet est facile, plus long sera le débat

La probabilité de dérive n'est nulle pour aucun fil, mais les sujets techniques simples sont un terreau très propice à l'égarement. Après tout, les sujets techniquement complexes ne seront bien compris que par une faible proportion de participants, certainement les développeurs les plus expérimentés ayant déjà pris part à ce genre de discussions des milliers de fois, et qui savent comment aboutir à un consensus viable pour tous.

Le consensus est donc plus difficile à atteindre sur des questions techniques faciles à comprendre (sur lesquelles on peut se faire aisément une opinion) ainsi que sur les sujets « légers » tels que l'organisation, la publicité, le financement, etc. Les gens peuvent participer à ces débats indéfiniment car il n'y a pas besoin de qualifications spécifiques pour le faire, ni de moyens clairs de savoir (même après coup) si la décision était bonne ou mauvaise, ou encore simplement parce qu'avoir les autres participants à l'usure est une tactique qui réussit parfois.

Le principe selon lequel la somme de discussions est inversement proportionnelle à la complexité du sujet a été énoncé il y a un moment déjà : il est connu officieusement sous le nom de l'effet *Bikeshed* [1]. Voici l'explication qu'en donne Poul-Henning Kamp dans un célèbre message aux développeurs BSD :

> *« C'est une longue histoire, ou plutôt une vieille histoire, mais très brève en réalité. C. Northcote Parkinson écrivit un livre dans les années 60 intitulé La Loi de Parkinson qui contient des opinions intéressantes sur la dynamique du management.*
>
> *[…]*
>
> *Dans cet exemple précis impliquant l'abri à vélos, l'autre élément important est une centrale nucléaire, je pense que cela illustre bien l'âge de l'ouvrage.*
>
> *Parkinson montre comment vous pouvez obtenir du bureau de direction l'accord de construction d'une centrale nucléaire coûtant plusieurs millions, voire un milliard de dollars, mais que, si vous voulez construire un abri à vélos, vous vous trouvez empêtré dans des discussions sans fin.*
>
> *Parkinson explique qu'une centrale nucléaire est si vaste, si chère et si complexe que les gens ne peuvent l'appréhender et, plutôt que d'essayer, ils préfèrent supposer qu'une autre personne a pris la peine de vérifier tous les détails avant d'aller plus loin. Richard P. Feynmann*

1. NdT : abri à vélos.

> *donne, dans son livre, quelques exemples intéressants et très pertinents concernant Los Alamos.*
>
> *À côté, nous avons l'abri à vélos. N'importe qui peut en construire un en un week-end, sans pour autant manquer le match à la télé. Aussi bien préparée et raisonnable que soit votre proposition, quelqu'un saisira cette opportunité pour montrer qu'il fait son boulot, qu'il veille au grain, qu'il est bien présent.*
>
> *Au Danemark, on appelle cela : laisser son empreinte. C'est une question d'amour-propre et de prestige, la fierté de pouvoir désigner quelque chose du doigt en disant : 'Voilà ! C'est moi qui l'ai fait'. C'est un trait de caractère très présent chez les politiciens, mais également chez toute personne à qui on donne un tout petit peu de pouvoir. Il n'y a qu'à voir les traces de pas dans le ciment frais. »*

(Son message complet vaut vraiment la peine d'être lu. Voir l'Annexe C.)

Toute personne qui a été membre d'un groupe de décision reconnaîtra ce dont parle Kamp. Cependant, il est généralement impossible de convaincre tout le monde d'éviter de peindre l'abri à vélos. La meilleure chose que vous puissiez faire est de signaler le phénomène quand il se produit, et convaincre les développeurs expérimentés, ceux dont les messages ont le plus de poids, de déposer leurs pinceaux avant les autres pour qu'eux au moins ne renforcent pas le bruit. Les manifestations pour-la-peinture-de-l'abri-à-vélo ne disparaîtront jamais complètement, mais on peut les rendre plus brèves et moins fréquentes en instillant la prise de conscience du phénomène dans la culture du projet.

Éviter les trolls

Un troll est une dispute portant souvent, mais pas toujours, sur une question plutôt insignifiante qui ne peut être résolue par la discussion car elle éveille des passions qui poussent les gens à continuer à argumenter jusqu'à faire prévaloir leur point de vue. Les trolls ne sont pas tout à fait semblables aux réunions-pour-la-peinture-de-l'abri-à-vélos. Les gens qui peignent les abris à vélos sont prompts à donner leur avis (car ils le peuvent), mais ne vont pas forcément le défendre avec véhémence, et émettent même à l'occasion d'autres opinions incompatibles pour montrer qu'ils prennent en compte tous les aspects de la question. Dans un sujet à troll, où prendre en compte les autres points de vue est un signe de faiblesse, tout le monde sait qu'il y a Une Réponse Juste, mais les participants ne sont simplement pas d'accord sur laquelle.

Une fois le troll lâché, il ne pourra pas être remis en cage sans que certains ne se sentent lésés. Inutile de déclarer en plein milieu d'un troll qu'il s'agit d'un troll. Tout le monde le sait déjà. Malheureusement, une caractéristique des trolls est la divergence d'opinion sur le règlement du conflit, la discussion ne peut donc aboutir ! De l'extérieur, il apparaît clairement qu'aucun des camps n'est en mesure de faire changer l'autre d'avis. De l'intérieur, la partie adverse est obtuse et n'a pas les idées claires, mais on peut en avoir raison à force d'intimidation. Attention : je ne dis pas qu'il n'y a pas de cause juste dans un troll. Parfois

il y en a une, et dans les trolls auxquels j'ai participé, c'était toujours la mienne évidemment. Mais peu importe, car il n'y a pas d'algorithme permettant de démontrer de manière convaincante qu'un camp a raison et l'autre tort.

Souvent, pour achever un troll, quelqu'un dira : « Nous avons déjà perdu trop de temps et d'énergie à discuter ! Ne voulez-vous pas simplement arrêter ? », mais ce n'est pas la bonne manière de faire. Il y a deux problèmes ici. D'abord, le temps et l'énergie perdus ne pourront plus être rattrapés, la question maintenant est de savoir l'effort qu'il reste à faire. Si certains pensent que poursuivre un peu la discussion amènera à la résolution du conflit, alors cela vaut encore la peine (de leur point de vue) de poursuivre.

L'autre problème quand on demande aux gens de laisser tomber, c'est que cela équivaut souvent à permettre à l'une des parties, celle du statu quo, de se déclarer victorieuse par forfait. Et dans certains cas, le statu quo n'est pas acceptable : tout le monde est d'accord, il faut prendre des décisions et agir dans un sens ou dans l'autre. Il vaut mieux poursuivre l'argumentation plutôt que d'abandonner le sujet. Mais comme le dilemme s'applique à tous de la même manière, il est aussi possible que la discussion continue éternellement.

Alors, comment gérer les trolls ?

Voici un début de réponse : faites en sorte qu'ils ne soient pas libérés. Ce n'est pas aussi irréalisable qu'il n'y paraît.

Vous pouvez anticiper certains trolls classiques : ils surgissent quand il est question de langages de programmation, de licences (voir la section intitulée « La GPL et la compatibilité de licence » du chapitre 9), de déguisement des adresses e-mail (voir la section « Le grand débat du Répondre à » du chapitre 3) et encore de quelques autres sujets. Chaque projet a également un ou deux trolls qui lui sont propres, avec lesquels les développeurs ayant de l'ancienneté se familiariseront rapidement. Les techniques pour stopper les trolls, ou pour en limiter les dégâts, sont à peu près les mêmes partout. Même si vous êtes certain que votre camp a raison, essayez de trouver un moyen de montrer votre ouverture et votre sympathie quand l'autre camp marque des points. Souvent dans les sujets à troll, chaque camp a construit des murs si hauts et tellement claironné que l'opinion adverse est pure folie, que le fait de céder ou de changer d'avis devient psychologiquement insoutenable : cela reviendrait à reconnaître non seulement que l'on a tort, mais que l'on a eu tort tout en étant sûr de soi. En montrant précisément que vous comprenez les arguments de la partie adverse, et que vous les trouvez sensés, même s'ils ne vous convainquent pas entièrement, vous rendrez cet aveu acceptable par l'autre camp. Faites un geste qui en appelle un autre en retour, et la situation s'améliorera. Même si vos chances d'obtenir le résultat technique souhaité n'en sont pas meilleures pour autant, vous aurez au moins évité des dommages collatéraux portant atteinte au moral du projet.

Lorsqu'un troll ne peut être évité, déterminez rapidement combien vous êtes prêt à sacrifier et soyez disposé à céder publiquement. Ce faisant, vous pouvez dire que vous reculez car la guerre ne vaut pas le coup, mais faites-le sans amertume et n'en profitez pas pour tirer une dernière salve sur les arguments du camp opposé. Le renoncement n'est efficace que s'il est fait de bonne grâce.

Les trolls sur les langages de programmation sont un peu spéciaux car, bien qu'ils impliquent souvent un haut degré de technicité, nombreux sont ceux qui se sentent qualifiés

pour y prendre part et les enjeux sont importants puisque de l'issue des débats dépendra le langage de l'essentiel du code du projet. Il vaut mieux choisir tôt le langage, en suivant l'avis des développeurs influents de départ, et le défendre pour la bonne et simple raison que c'est celui avec lequel vous êtes tous plus à l'aise, et non parce qu'il est meilleur qu'un autre remplaçant. Ne laissez jamais la conversation dégénérer en une comparaison académique entre langages de programmation (ceci semble se produire plus généralement quand quelqu'un évoque Perl, pour je ne sais quelle raison) : c'est une impasse dans laquelle vous devez refuser de vous laisser entraîner.

Pour plus de références historiques aux trolls, voir The Jargon File [1], ainsi que l'article de Danny Cohen [2] qui a rendu ce terme populaire.

L'effet « minorité bruyante »

Sur les listes de discussion, il est très facile pour une minorité de donner l'impression qu'il y a de grandes divergences en noyant la liste sous une avalanche de longs messages. C'est un peu une technique de pirate, une minorité de blocage, si ce n'est que l'illusion d'un désaccord diffus est encore plus puissante, car elle est disséminée à travers un nombre arbitraire de messages discrets, et que la plupart ne se donneront pas la peine de suivre à la trace qui a dit quoi et quand. Ils auront juste l'impression instinctive que le sujet est controversé, et attendront que le soufflé retombe.

La meilleure façon de contrer cet effet est de montrer très clairement, preuves à l'appui, à quel point le groupe en désaccord est petit comparé à tous ceux qui sont d'accord. Pour rendre la disproportion plus évidente, vous pourrez sonder en privé ceux qui ont gardé le silence mais que vous supposez en accord avec la majorité. Ne dites rien qui puisse faire croire que les dissidents essayaient de gonfler la portée de leur impact. Ce n'était pas nécessairement leur but, et, même si c'était le cas, il n'y a aucun avantage stratégique à le faire remarquer. Il vous suffit de mettre en avant les chiffres réels pour que les participants réalisent que leur perception de la situation ne correspond pas à la réalité.

Ce conseil ne s'applique pas qu'aux problèmes où les camps sont bien marqués. Il s'applique à toutes les discussions faisant couler beaucoup d'encre virtuelle mais dont le sujet n'est pas nécessairement un vrai problème aux yeux du groupe. Après un certain temps, si vous convenez que le sujet ne mérite pas d'action, et que vous voyez qu'il n'avance pas (même s'il a généré beaucoup d' e-mails), vous pouvez simplement faire observer publiquement qu'il n'y a pas de progrès. Si l'effet « minorité bruyante » s'était mis en place, votre message ressemblera à une bouffée d'air frais. La perception générale de la discussion ne sera plus aussi nette : « Heu, c'est vrai qu'on dirait que c'est important parce qu'il y a beaucoup de messages, mais je n'ai pas l'impression qu'on progresse ». En expliquant que le déroulement de la discussion l'a fait paraître plus animée qu'elle ne l'est vraiment, vous en donnez rétrospectivement une nouvelle image qui permet aux participants d'avoir un point de vue dépassionné sur ce qui s'en dégage.

1. http ://catb.org/%7Eesr/jargon/html/H/holy-wars.html
2. http ://www.ietf.org/rfc/ien/ien137.txt

3. Les personnes difficiles

Il n'est pas plus facile de traiter avec les gens difficiles sur les forums électroniques que dans la vraie vie. Par « difficiles », je ne veux pas dire « grossiers ». Les gens grossiers sont embêtants, mais pas nécessairement difficiles. Nous avons déjà abordé dans cet ouvrage la manière de s'en occuper : faites-leur remarquer leur grossièreté une première fois, à partir de quoi ignorez-les ou traitez-les comme n'importe quelle autre personne. S'ils continuent à être grossiers, ils se rendront si impopulaires qu'ils n'auront plus d'influence sur les autres au sein du projet ; le problème qu'ils posent se résoudra ainsi de lui-même.

Les cas réellement difficiles sont ceux qui, n'étant pas ouvertement grossiers, manipulent les autres ou abusent des processus du projet de telle sorte qu'ils monopolisent le temps et l'énergie des autres sans rien apporter de positif au projet. Ces personnes cherchent souvent les points d'achoppement des procédures mises en place, et s'en servent comme levier pour obtenir plus d'influence qu'ils n'en auraient autrement. C'est bien plus insidieux que de la simple grossièreté, car ni le comportement ni les dommages qu'ils causent ne sont visibles pour un observateur occasionnel. Un exemple classique est celui du bloqueur, celui (à l'air on ne peut plus raisonnable bien sûr) qui ne cesse de répéter que la réflexion n'est pas encore mûre pour qu'on prenne une décision, qui propose toujours plus de solutions ou de nouvelles approches alors qu'au fond il sent que le consensus ou le vote est en train de prendre forme et qu'il n'aime pas la direction prise. Un autre exemple est celui du débat qui n'aboutit pas à un consensus, mais dans lequel le groupe essaie au moins de clarifier les points de désaccord, et de produire un compte-rendu auquel chacun pourra se référer par la suite. L'obstructionniste, qui sait que le compte-rendu peut avoir une issue qui lui déplaît, essaiera de le reporter en compliquant sans cesse la question de ce qui devrait y figurer, soit en s'opposant aux suggestions raisonnables, soit en introduisant de nouveaux points inattendus.

Gérer les personnes difficiles

Pour contrer ce genre de comportements, il est utile de comprendre la mentalité de ceux qui y recourent. De manière générale, ils ne le font pas consciemment. Personne ne se réveille le matin en se disant : « Tiens, aujourd'hui je vais manipuler cyniquement les procédures pour faire mon obstructionniste agaçant ». En revanche, de telles actions sont souvent précédées d'un sentiment semi-paranoïaque de mise à l'écart des échanges et des décisions du groupe. La personne a l'impression d'être déconsidérée ou, dans les cas plus graves, se figure qu'il y a une conspiration contre elle où les autres membres du projet créent un club fermé dont elle est exclue. Ce qui justifie, à ses yeux, de prendre le règlement à la lettre et de s'engager dans une manipulation formelle des procédures du projet, afin que les autres la prennent au sérieux. Dans les cas extrêmes, la personne peut même croire qu'elle se bat seule contre tous pour sauver le projet lui-même.

Tout le monde ne remarquera pas ces attaques de l'intérieur au même moment, c'est dans leur nature même, et certains ne les verront que si on leur en fournit des preuves solides. Par conséquent, neutraliser ces attaques peut demander pas mal d'efforts. Il ne suffit pas de vous convaincre de ce qui est en train de se produire : il vous faudra aligner des preuves suffisantes et les présenter de manière réfléchie pour convaincre les autres.

Ces combats demandent beaucoup d'énergie, votre meilleure attitude sera peut-être de tolérer momentanément ces comportements. C'est un peu comme une maladie parasitaire bénigne : si elle n'affaiblit pas trop le projet, on peut la supporter, les médicaments pourraient avoir des effets secondaires négatifs. Cependant, si la tolérance cause trop de dommages, il est alors temps d'agir. Commencez à rassembler des notes sur les faits que vous observez. Assurez-vous d'y inclure des références à des archives publiques, c'est là une des raisons pour lesquelles le projet garde des traces, vous pouvez donc les utiliser. Lorsque vous aurez bien documenté l'affaire, entamez une phase d'échanges privés avec d'autres participants. Ne leur dites pas ce que vous avez observé : recueillez d'abord leurs propres impressions. C'est sans doute votre dernière chance d'avoir un retour non filtré sur leur perception de la situation : une fois que vous commencerez à en parler ouvertement, les avis seront polarisés et personne ne se souviendra de son avis initial sur la question.

Si les discussions privées montrent que vous n'êtes pas seul, il est temps de faire quelque chose. C'est là qu'il faut être vraiment prudent, car il est très facile pour ce genre de personnes d'essayer de faire croire qu'on leur tombe dessus injustement. Quoi que vous fassiez, ne les accusez jamais de détourner les procédures du projet de manière malintentionnée, d'être paranoïaques, ni de toute autre chose que vous soupçonnez être vraie. Votre stratégie doit être de vous montrer à la fois raisonnable et concerné par le bien-être global du projet, avec pour objectif soit de modifier le comportement de cette personne, soit de lui faire quitter le projet. Selon les autres développeurs et vos relations avec eux, il peut être avantageux de réunir d'abord des alliés en privé… ou non : cela pourrait créer des réticences en coulisses si l'on pense que vous vous lancez dans une campagne de rumeurs abusives.

Souvenez-vous que, bien que la personne « difficile » soit celle qui agit de manière destructrice, c'est vous qui paraîtrez destructeur si vous lancez des accusations publiques sans pouvoir les étayer. Soyez certain d'avoir de nombreux exemples pour démontrer ce que vous avancez, et dites-le aussi gentiment que possible tout en étant direct. Vous ne persuaderez peut-être pas la personne en question, ce qui n'est pas très important du moment que vous arrivez à convaincre les autres.

Étude de cas

Je ne me souviens que d'une seule fois, au cours des dix années passées dans le logiciel libre, où les choses sont arrivées à un point où nous avons dû demander à quelqu'un de cesser de poster des messages. Comme c'est souvent le cas, il n'était pas grossier et voulait vraiment être utile. Mais il ne savait pas quand il fallait poster et quand il ne le fallait pas. Nos listes étaient ouvertes au public et il postait si souvent pour poser des questions sur tellement de sujets qu'il devenait une nuisance pour la communauté. Nous avions déjà essayé de lui demander gentiment de chercher un peu plus les réponses avant de poster, sans résultat.

La stratégie qui finit par payer est un exemple parfait de l'utilisation de données neutres, quantitatives pour monter un dossier solide. Un de nos développeurs alla fouiller dans les archives, puis envoya le message suivant en privé à quelques développeurs. Le contrevenant (le troisième nom dans la liste ci-dessous, apparaissant ici comme « A. Nonyme ») avait très peu de relations avec le projet, il n'avait contribué ni au code ni à la documentation. Et pourtant, parmi les participants, il était troisième en nombre de messages envoyés sur les listes :

> De : "Brian W. Fitzpatrick" <fitz@collab.net> À : [... liste des destinataires omise pour conserver l'anonymat] Sujet : The Subversion Energy Sink Date : Wed, 12 Nov 2003 23 :37 :47 -0600 Dans les 25 derniers jours, les 6 personnes ayant posté le plus dans la liste svn [dev|users] sont :
>
> 294 kfogel@collab.net 236 "C. Michael Pilato" <cmpilato@collab.net> 220 "A. Nonyme" <anonyme@problematicposter.com> 176 Branko Čibej <brane@xbc.nu> 130 Philip Martin <philip@codematters.co.uk> 126 Ben Collins-Sussman <sussman@collab.net>
>
> Je dirais que 5 de ces personnes contribuent à atteindre Subversion 1.0 dans un avenir proche.
>
> J'ajouterai que l'une de ces personnes prend du temps et de l'énergie aux 5 autres, sans parler de la liste dans son ensemble, ralentissant ainsi (fût-ce involontairement) le développement de Subversion. Je n'ai pas fait une analyse fil par fil, mais un vgrepp sur mon spool d'e-mail Subversion me dit que pour chaque e-mail de cette personne, au moins deux des cinq autres personnes de la liste ci-dessus prennent le temps de lui répondre.
>
> Je pense qu'une intervention musclée est nécessaire, quitte à effrayer la personne mentionnée plus haut. La délicatesse et la gentillesse se sont avérées sans effet.
>
> dev@subversion est une liste destinée à faciliter le développement d'un logiciel de gestion de versions, ce n'est pas une séance de thérapie de groupe.
>
> -Fitz, qui cherche à ne pas se noyer dans trois jours d'e-mails svn qu'il a laissés s'accumuler.

Bien qu'il n'en eût pas l'air au début, le comportement de A. Nonyme était un cas classique d'abus des procédures. Il ne faisait rien de flagrant comme retarder un vote, mais il tirait profit de la politique d'auto-modération de la liste de diffusion. C'était à chaque membre de juger quand et sur quoi poster. Donc, nous n'avions pas de procédure de recours pour faire face à une personne qui n'avait pas ou n'exerçait pas cette capacité de discernement. On ne pouvait pas dire que ce gars violait des règles, mais pourtant tout le monde savait que ses messages fréquents devenaient un problème sérieux.

La stratégie de Fitz s'est avérée, rétrospectivement, excellente. Il a réuni des preuves accablantes, mais les a diffusées discrètement, les communiquant d'abord aux quelques personnes dont le soutien s'avérait décisif pour une action drastique. Ils étaient d'accord sur le fait que des mesures étaient nécessaires, et finalement nous avons appelé A. Nonyme au téléphone, nous lui avons décrit le problème directement et lui avons demandé de cesser de poster. Il n'en a jamais vraiment compris les raisons : s'il avait été en mesure de les comprendre, il aurait probablement fait preuve de discernement dès les début. Mais il accepta

de ne plus poster et la liste devint utilisable à nouveau. Un des motifs pour lesquels cette stratégie a fonctionné était peut-être la menace implicite que nous pourrions mettre en place un filtrage de ses messages *via* le logiciel de modération couramment utilisé pour prévenir le spam (voir la section « Se prémunir du Spam » dans le chapitre 3). Mais si nous avions cette option en réserve, c'est parce que Fitz avait d'abord réuni le soutien nécessaire de personnages-clés.

4. Gérer la croissance

Le prix du succès est lourd dans le monde de l'Open Source. Au fur et à mesure que votre logiciel devient populaire, le nombre de personnes cherchant des renseignements augmente de manière très importante tandis que le nombre de personnes capables de fournir ces renseignements croît bien plus lentement. De plus, même si la proportion entre les deux était à peu près équilibrée, il resterait encore un problème fondamental : la gestion de la communication. Prenons les listes de diffusion par exemple. La plupart des projets disposent d'une liste pour les questions générales des utilisateurs, cette liste s'appelle parfois « utilisateurs », « discussion », « aide » ou quelque chose comme ça. Quel qu'en soit le nom, le but de cette liste est toujours le même : fournir un lieu où certains trouvent des réponses à leurs questions, tandis que d'autres observent et (normalement) s'imprègnent des connaissances en suivant ces échanges.

Pour mettre en relation quelques milliers d'utilisateurs ou pour gérer une grosse centaine de messages quotidiens, ces listes de diffusion sont très efficaces. Mais au-delà, le système montre ses limites car chaque abonné reçoit chaque message. Si le nombre d'e-mails envoyés excède le nombre de messages qu'une personne peut traiter en un jour, la liste perd toute son utilité. Imaginez, par exemple, que Microsoft décide de créer une liste de diffusion pour Windows 7, dont les utilisateurs se comptent par centaines de millions. Si chaque jour seulement un millième d'entre eux envoyait une question à cette liste, elle croulerait sous des milliers de messages quotidiens ! Une telle liste ne pourrait jamais exister évidemment, personne n'y resterait abonné. Mais ce problème ne concerne pas que les listes de diffusion : le même raisonnement s'applique aux canaux IRC, aux forums ou à tout système mettant les utilisateurs en relation directe avec le projet. Les implications en sont inquiétantes : le modèle classique d'assistance massivement parallélisée ne s'adapte pas aux projets partant à la conquête du monde.

Il n'y a pas d'explosion quand les forums atteignent leur point culminant. Il se produit seulement une suite d'événements ponctuels néfastes : les gens se désabonnent des listes, ou désertent le canal IRC, ou cessent d'une manière ou d'une autre de se donner la peine de poser des questions car ils comprennent qu'ils ne seront pas entendus au milieu de tout ce bruit. Comme de plus en plus de personnes font ce choix bien compréhensible, l'activité du forum semblera se maintenir à un niveau gérable. Mais elle reste gérable justement parce que les gens rationnels (ou du moins expérimentés) ont commencé à chercher l'information ailleurs, tandis que les gens inexpérimentés restent et continuent à poster. En d'autres termes, quand on continue à utiliser des modèles de communication non modulables alors que le projet grandit, la qualité moyenne des questions et des réponses tend à baisser, ce qui donne l'impression que les nouveaux utilisateurs sont plus bêtes que leurs prédécesseurs,

alors qu'en fait ce n'est probablement pas le cas. C'est simplement que la rentabilité de ces forums à grande audience est plus basse, donc naturellement ceux qui en ont l'expérience chercheront ailleurs leurs réponses en priorité. Ajuster les mécanismes de communication pour faire face à la croissance du projet implique deux stratégies liées :

1. Identifier les parties spécifiques d'un forum qui ne subissent pas une croissance débridée, contrairement au reste du forum, et les détacher dans un nouveau forum plus spécialisé (c.-à-d. : ne laissez pas le mauvais tirer le bon vers le bas).
2. S'assurer que plusieurs sources d'information automatisées sont à la disposition des utilisateurs, qu'elles sont bien organisées, mises à jour et faciles à trouver.

La stratégie n°1 est généralement assez facile à mettre en œuvre. La plupart des projets démarrent avec un forum principal : une liste de discussion générale dans laquelle les propositions de fonctionnalités, les questions relatives à la conception et les problèmes liés au code sont abordés en vrac. Tous les participants y sont inscrits. Au bout d'un certain temps, il devient évident que la liste a évolué en plusieurs sous-listes orientées chacune sur un sujet. Par exemple, certains fils concernent clairement le développement et la conception : d'autres sont des questions d'utilisateurs sur le fonctionnement « Comment faire X ? » : un troisième groupe porte peut-être sur le traitement des rapports de bogue et les demandes d'améliorations, etc. Un individu donné peut bien sûr prendre part à plusieurs types de fils différents, mais il faut veiller à ce que ces catégories ne se recoupent pas trop. Les fils pourraient être répartis dans des listes séparées sans causer un cloisonnement nuisible car les ils traversent rarement la frontière des sujets.

Procéder à cette division est un processus en deux étapes. On crée d'abord la nouvelle liste (ou canal IRC, ou autre), puis on passe le temps nécessaire à gentiment harceler les gens et leur rappeler l'utilisation appropriée des nouveaux forums. Cette dernière étape peut durer des semaines, mais un jour les utilisateurs s'y feront. Vous devez simplement vous astreindre à prévenir systématiquement l'expéditeur quand son message est posté au mauvais endroit, et le faire de manière visible, pour que les autres soient encouragés à vous relayer dans l'aiguillage. Il est utile également d'avoir une page Web avec un index de toutes les listes disponibles : vos réponses peuvent simplement renvoyer à cette page et, en prime, le destinataire aura appris à se reporter au guide d'utilisation avant de poster.

La stratégie n°2 est un processus à long terme, qui dure tant que dure le projet, et implique plusieurs participants. Bien sûr, il est question ici de documentation à jour (voir la section « Documentation » du chapitre 2) et d'orientation du public. Mais c'est bien plus que cela. La section qui suit examine cette stratégie en détail.

Utilisation visible des archives

Traditionnellement, toutes les communications échangées dans un projet Open Source (exceptées parfois les conversations IRC) sont archivées. Les archives sont publiques, consultables et ont une stabilité référentielle : c'est-à-dire qu'une information enregistrée a une adresse donnée où elle reste pour toujours.

Utilisez ces archives autant que possible et aussi visiblement que possible. Même si vous connaissez par cœur la réponse à une question, si vous pensez qu'il y a dans les archives

une référence qui contient la réponse, prenez le temps d'aller la chercher et de la montrer. Chaque fois que vous faites ceci publiquement et de manière visible, quelqu'un apprend pour la première fois que les archives sont là et qu'elles contiennent des réponses aux questions. De même, en vous référant aux archives plutôt que de ré-écrire la réponse, vous renforcez la règle sociale qui lutte contre la duplication de l'information. Pourquoi avoir la même réponse à deux endroits différents ? Les gens retrouveront plus facilement une réponse qu'ils ont déjà dénichée si celle-ci n'est pas répliquée x fois. Des références faciles à localiser contribuent également à la qualité des résultats des recherches en général car elles renforcent la classification des ressources ciblées par les moteurs de recherche Internet.

Il existe pourtant des cas où dupliquer l'information peut avoir un sens. Imaginez par exemple qu'il existe déjà une réponse dans les archives, où un autre contributeur que vous dit :

> Il s'avère que vos index de Scanley ont été embrouillés. Pour les désembrouiller, suivez les étapes suivantes : 1. Éteignez le serveur Scanley ; 2. Lancez le programme "désembrouillage" livré avec Scanley ; 3. Redémarrez le serveur.

Puis, des mois plus tard, vous verrez un autre message expliquant que les index de quelqu'un ont été embrouillés. Vous cherchez dans les archives et trouvez l'ancienne réponse ci-dessus, mais vous vous rendez compte qu'il y manque quelques étapes (soit par erreur, soit parce que le logiciel a changé depuis la rédaction du message). Il vaut alors mieux poster de nouvelles instructions, plus complètes, et de rendre explicitement obsolète le message précédent en le mentionnant :

> Il s'avère que vos index de Scanley ont été embrouillés. Nous avons vu ce problème en juillet dernier et A. Nonyme avait posté une solution à l'adresse http ://blablabla.bla. Vous trouverez ci-dessous des indications plus complètes pour désembrouiller vos index, basées sur celles de A. Nonyme mais un peu plus développées : 1. Éteignez le serveur Scanley ; 2. Devenez l'utilisateur habituellement connecté au serveur Scanley ; 3. Sous ce nom d'utilisateur, lancez le programme "désembrouillage" dans les index ; 4. Lancez Scanley à la main pour voir si les index fonctionnent ; 5. Redémarrez le serveur.

(Dans un monde idéal il serait possible d'attacher une note à l'ancien message précisant que des informations plus récentes ont été apportées et d'y inclure un lien vers le nouvel article. Cependant, je ne connais aucun logiciel d'archivage qui propose une fonction « rendu obsolète par », peut-être parce qu'il serait difficile à mettre en œuvre de façon à ce que l'intégrité des archives soit préservée en tant que trace verbatim. C'est là une autre raison qui justifie la création de pages Web dédiées consignant les réponses aux questions courantes.)

Les archives sont le plus souvent utilisées pour chercher des réponses aux questions techniques, mais leur importance pour le projet va au-delà. Si les règles formelles d'utilisation du projet constituent la loi ordinaire, les archives représentent le droit coutumier : des archives

de toutes les décisions prises et des discussions qui y ont mené. Dans toute discussion récurrente, il est plus ou moins obligatoire de nos jours de commencer par une recherche dans les archives. Ceci permet de commencer la discussion par un résumé de l'état courant des choses, d'anticiper des objections, de parer aux réticences et, probablement, de trouver des idées auxquelles vous n'auriez pas pensé. Les autres participants aussi estimeront que vous êtes censé avoir fait une recherche dans les archives. Même si les discussions précédentes n'ont mené nulle part, vous devriez mettre un lien qui y renvoie, au moment de remettre la question sur le tapis, pour que les intervenants constatent par eux-mêmes que : a) elles n'ont mené nulle part et b) vous avez fait votre boulot. Ils seront alors en mesure de dire quelque chose qui n'a pas été dit auparavant.

Traitez toutes les ressources comme des archives

Tous les conseils précédents ne s'appliquent pas qu'aux archives des listes de diffusion. Certaines informations doivent être enregistrées à des adresses stables, faciles à trouver. Ceci devrait être un principe en vigueur pour toute information concernant le projet. Prenons la FAQ du projet comme cas d'étude.

Comment une FAQ est-elle utilisée ?

1. Les utilisateurs veulent pouvoir y chercher des mots et des phrases.
2. Ils veulent pouvoir la parcourir, aller à la pêche aux informations sans chercher nécessairement de réponses à des questions particulières.
3. Ils s'attendent à ce que son contenu soit accessible aux moteurs de recherche tel que Google, de manière à ce que les recherches pointent vers les rubriques de la FAQ.
4. Ils veulent pouvoir indiquer à d'autres personnes des articles spécifiques.
5. Ils veulent pouvoir y ajouter de nouveaux éléments. Notez néanmoins que ceci est bien moins courant que la recherche de réponses : on lit plus souvent une FAQ qu'on n'y écrit.

Le point 1 implique que la FAQ soit disponible sous forme de texte. Les point 2 et 3 impliquent qu'elle doive être disponible au format HTML, et le point 2 indique aussi que le HTML doit être adapté à la lecture (vous veillerez donc à sa lisibilité et à son aspect) et proposer une table des matières. Le point 4 signifie que chaque entrée de la FAQ doit faire l'objet d'une ancre à son nom, un repère qui permet d'atteindre un endroit particulier de la page. Le point 5 suppose que les fichiers sources de la FAQ doivent être accessibles de manière commode (voir la section « Tout versionner » du chapitre 3), dans un format facile à éditer.

Ancres nommées (named anchors) et attributs ID
Il existe deux solutions pour qu'un navigateur se place à un endroit précis d'une page Web : les ancres nommées et les attributs ID.
Une ancre nommée n'est qu'une ancre html normale (<a>…), à laquelle on attribue un « nom » :

...
Des versions plus récentes de HTML supportent un attribut id générique qui peut être attaché à n'importe quel élément HTML et pas seulement à <a>. Par exemple :
* ... *
Ancres nommées et attributs id sont utilisés de la même façon. On fait suivre l'URL d'un dièse et de l'étiquette pour que le navigateur aille directement à cet endroit de la page :
http ://projet.exemple.com/faq.html#monetiquette
Théoriquement tous les navigateurs gèrent les ancres nommées ; les plus récents gèrent aussi les attributs id. Par prudence, je recommande soit de n'utiliser que des ancres nommées, soit des ancres nommées et des attributs id ensemble (avec la même étiquette pour les deux, évidemment). Les ancres nommées ne sont pas auto-fermantes : même s'il n'y a pas de texte à l'intérieur de l'élément, il faut l'écrire en deux parties :
* *
. . . bien que normalement il devrait y avoir un texte, comme un titre de section.
Que vous utilisiez une ancre nommée, un attribut id ou les deux, rappelez-vous que les visiteurs parcourant la page sans utiliser d'étiquette ne voient pas la différence. Mais cette personne peut vouloir connaître l'étiquette d'une section en particulier, pour pouvoir envoyer l'URL de la réponse de la FAQ à un ami par exemple. Pour l'y aider, ajoutez un titre aux éléments auxquels vous avez ajouté un « nom » et/ou un « id », par exemple :
...
En plaçant le pointeur de la souris sur le texte qui se trouve à l'intérieur de l'attribut « title », la plupart des navigateurs ouvriront une petite fenêtre où s'affiche le titre. J'ajoute généralement le dièse pour rappeler à l'utilisateur que c'est le symbole qu'il devra ajouter à la fin de l'URL pour aller directement à cet endroit la prochaine fois.

Ce n'est qu'un exemple de mise en page de la FAQ qui permet d'en faire un ressource présentable. Les mêmes propriétés, accès direct lors de la recherche, accessibilité aux principaux moteurs de recherche Internet, navigabilité, stabilité du référentiel et le cas échéant, capacité d'édition s'appliquent également à d'autres pages Web, à l'arborescence du code source, au système de suivi de bogues, etc. Il se trouve que la plupart des logiciels d'archivage des listes de diffusion ont reconnu depuis longtemps l'importance de ces propriétés, c'est pourquoi les listes ont tendance à inclure ces fonctions de manière native alors que d'autres formats exigent un effort supplémentaire de la part de celui qui est chargé de leur maintenance (le chapitre 8 examine comment répartir la charge de cette maintenance entre plusieurs volontaires).

Codifier la tradition

Quand l'histoire d'un projet s'étoffe et gagne en complexité, la quantité de données que chaque nouvel arrivant doit assimiler augmente. Ceux qui participent au projet depuis longtemps ont pu apprendre, voire forger, les conventions du projet en cours de route. Très souvent, ils n'auront pas clairement conscience du corpus de traditions qui s'est accumulé, et seront surpris des nombreux impairs que les nouveaux sont susceptibles de commettre. Bien sûr, le problème n'est pas que les nouveaux venus soient moins bons qu'avant : c'est que

l'effort d'acclimatation qu'ils doivent consentir est plus important que pour ceux qui les ont précédés.

Les traditions qu'un projet accumule portent aussi bien sur la communication et la préservation de l'information que sur la qualité du code ou d'autres détails techniques. Nous avons déjà examiné ces deux cas de figure et des exemples ont été fournis, respectivement dans les sections « Documentation développeur » du chapitre 2 et « Tout mettre par écrit » du chapitre 4. Ici, nous allons plus particulièrement nous concentrer sur la mise à jour de ces informations avec l'évolution du projet, et, plus particulièrement, celles relatives à la gestion des communications : c'est en effet le domaine qui évolue le plus quand le projet gagne en taille et en complexité.

Premièrement, cherchez ce qui est le plus déroutant pour la communauté. Si les mêmes situations se répètent sans cesse, particulièrement avec les nouveaux participants, il est probable que le guide n'est pas documenté. Deuxièmement, ne vous lassez pas de répéter cent fois les mêmes choses, et n'ayez pas l'air las en les répétant. Vous aurez, ainsi que les autres vétérans du projet, à vous répéter souvent : c'est un effet secondaire inévitable de l'arrivée de nouveaux participants.

Chaque page Web, chaque message de la liste, chaque canal IRC doit être considéré comme un espace publicitaire : non pas pour passer des annonces commerciales, mais pour faire la réclame des ressources propres au projet. Ce que vous mettrez dans chacun de ces espaces dépend de la population susceptible de le lire. Un canal IRC consacré aux questions des utilisateurs, par exemple, amènera des gens qui n'ont jamais été en relation avec le projet auparavant, typiquement quelqu'un qui vient d'installer le logiciel et qui aimerait une réponse immédiate à sa question (après tout, si ça pouvait attendre, il l'aurait postée sur la liste de diffusion, ce qui lui prendrait probablement moins de temps dans l'ensemble, bien que le délai pour la réponse soit plus long). Les gens n'investissent généralement pas le canal IRC sur la durée : ils surgissent, posent leur question et s'en vont.

C'est pourquoi le sujet du canal doit viser les gens qui cherchent des réponses techniques immédiates, plutôt que des gens qui pourraient s'investir dans le projet à long terme et pour qui le manuel des usages de la communauté serait plus approprié. Voici comment un canal réellement actif gère la question (voir aussi l'exemple dans la section « IRC et les chats en temps réel » du chapitre 3) :

> Vous êtes actuellement sur #linuxhelp. Le sujet de #linuxhelp est SVP LISEZ http ://www.catb.org/ esr/faqs/smart-questions.html et http ://www.tldp.org/docs.html#howto AVANT de poser une question | les règles du canal se trouvent ici : http ://www.nerdfest.org/ lh_rules.html | Veuillez consulter http ://-kerneltrap.org/node/view/799 avant de poser une question sur la mise à jour 2.6.8.1 ou 2.4.27 | sinistre algorithme de hachage : http ://tinyurl.com/6w8rf | reiser4 out

Dans les listes de diffusion, l'« espace publicitaire » c'est la note de bas de page (*footer*) ajoutée à la fin de chaque message. La plupart des projets y font figurer les instructions pour s'inscrire/se désinscrire ainsi qu'éventuellement un lien vers la page d'accueil ou vers

la FAQ. Vous pensez peut-être que tous les abonnés de la liste savent où trouver ces informations, et c'est vraisemblablement le cas, mais bien des personnes autres que les abonnés voient les messages de la liste. Des liens peuvent pointer vers le courrier archivé en de nombreux endroits : de fait, certains messages deviennent si connus qu'ils peuvent avoir plus de lecteurs hors-liste que dans la liste.

Le formatage peut apporter un réel avantage. Par exemple, dans le projet Subversion, nous avions un succès limité avec l'utilisation de la technique de filtrage de bogues décrite dans la section « Filtrer le système de suivi des bogues en amont » du chapitre 3. De nombreux rapports de bogues bidons étaient créés par des gens inexpérimentés et, chaque fois que cela se produisait, il fallait former le rapporteur exactement de la même manière que les 500 rapporteurs précédents. Un jour, alors qu'un de nos développeurs avait fini par perdre patience, et s'en était pris à un pauvre utilisateur qui n'avait pas lu assez attentivement le manuel du traqueur de bogues, un autre développeur décida que la situation avait assez duré. Il proposa une refonte de la page d'accueil du traqueur de bogues pour faire ressortir (en gros caractères rouges et gras sur fond jaune, bien centrés en tête de page) le point le plus important, à savoir l'incitation à discuter du bogue sur la liste de diffusion et sur le canal IRC avant de remplir un rapport. Ce qui fut fait (vous pouvez voir le résultat sur subversion.tigris.org [1]). Il en résulta une baisse notable du nombre de rapports de bogues bidons. Nous en recevons encore bien sûr, nous en recevrons toujours, mais le taux a baissé considérablement, et ce malgré l'augmentation du nombre d'utilisateurs. Ainsi, non seulement notre base de données contient moins de déchets, mais ceux qui travaillent à résoudre les bogues conservent leur bonne humeur et ont plus de chance de rester aimables au moment de répondre à l'un de ces rares rapports bidons. L'image du projet n'en est que meilleure et la santé mentale des volontaires est épargnée.

Conclusion : il ne suffit pas simplement de mettre les règles par écrit. Il faut également les rendre visibles à ceux qui en ont le plus besoin, et les présenter de telle sorte que leur rôle de support d'introduction au projet soit suffisamment clair pour ceux qui ne le connaissaient pas encore.

Des pages Web statiques ne sont pas les seuls espaces pour promouvoir les us et coutumes du projet. Une certaine dose de veille interactive (dans le sens de « rappel amical » plutôt que de « mise à l'amende ») est également nécessaire. Toute révision par les pairs, y compris les révisions de commits décrites dans la section intitulée « Effectuez une inspection visible du code » du chapitre 2, devraient commenter le respect des normes du projet, notamment des conventions de communication.

Voici un autre exemple tiré du projet Subversion : nous avons fixé par convention que « r12908 » signifiait « révision 12908 dans le dépôt de gestion de versions ». Le préfixe en bas de casse « r » est facile à taper et sa taille étant moitié moindre que celle des chiffres on obtient un bloc de texte facilement reconnaissable. Bien sûr, le fait de fixer cette convention n'implique pas que tout le monde va l'utiliser immédiatement et uniformément. Ainsi, quand arrive un *commit* avec un message du fichier journal tel que celui-ci :

> ——————————————————————- r12908 | qsimon | 2005-02-02 14 :15 :06 -0600 (Wed, 02 Feb 2005) | 4 lines Correc-

1. http ://subversion.tigris.org/project_issues.html

> tif de A. Nonyme <anonyme@gmail.com> * trunk/contrib/client-side/psvn/psvn.el : Corrigé quelques typos sur la révision 12828 —
> ——————————————————————-

… la relecture de ce *commit* comprend également un petit mot du genre : « Au fait, utilisez de préférence 'r12828' au lieu de 'révision 12828' pour parler des changements précédents. » Ce n'est pas juste de la pédanterie : c'est aussi important pour l'indexation automatique que pour la lisibilité humaine.

Il devrait y avoir une méthode de référence pour des unités communes utilisées partout uniformément, des normes que le projet puisse exporter de manière effective. Elles permettent aux gens d'écrire des outils qui rendent plus facile l'exploitation des échanges du projet. Une révision sous la forme « r12828 » peut être transformée en lien vivant dans le système de navigation du dépôt par exemple. Ceci serait plus difficile à faire si la révision avait été notée sous la forme « révision 12828 », d'une part parce que cette expression aurait pu être divisée par un retour à la ligne, d'autre part parce qu'elle est moins distincte (le mot révision, avec ou sans accent, apparaîtra souvent isolé des chiffres, tandis que la combinaison « r12828 » ne peut que signifier « numéro de révision »). Des problèmes similaires se posent pour les numéros des bogues, les points de la FAQ (une piste : utilisez des URL avec des ancres nommées, comme indiqué dans « Ancres nommées et attributs ID »), etc.

Même pour les unités où il n'existe pas d'étalon défini, les gens devraient être encouragés à fournir les informations-clés de manière uniformisée. Par exemple, pour vous référer à un message d'une liste de diffusion, ne mentionnez pas seulement l'émetteur et le sujet, citez également l'URL de l'archive et le Message-ID de l'en-tête. Ce dernier permet à ceux qui ont leur propre copie de la liste de diffusion (certaines personnes conservent une copie hors ligne, pour l'utiliser sur un portable lors d'un voyage par exemple) d'identifier univoquement le message même s'ils n'ont pas accès aux archives. L'émetteur et le sujet ne suffisent pas car la même personne peut avoir posté plusieurs fois le même jour dans le même fil.

Plus un projet grandit plus cette uniformité devient importante, et quelle que soit la situation, les mêmes règles sont appliquées, ce qui incite les acteurs à les suivre. Ceci, en retour, réduit le nombre de questions qu'ils ont à poser. Avoir 1 lecteur ou 1 million de lecteurs ne fait pas de différence : les problèmes d'échelle commencent à apparaître seulement quand un certain pourcentage de ces lecteurs posent des questions. Quand le projet grandit, il faut donc réduire ce pourcentage en augmentant la densité et l'accessibilité de l'information pour que chaque personne ait plus de chance de trouver ce dont elle a besoin sans avoir à le demander.

5. Pas de discussion dans le système de suivi de bogues

Dans tous les projets qui font un usage actif d'un système de suivi de bogues existe le danger que celui-ci devienne un forum de discussion en lui-même, malgré la présence des listes de diffusion. Généralement cela commence innocemment : quelqu'un note un problème et propose une solution ou un correctif partiel. Quelqu'un d'autre le remarque, se rend compte que la solution proposée n'a pas que du bon et attache une autre note indiquant ces problèmes. La première personne répond en ajoutant une note sur la question… et ainsi de suite.

Un constat s'impose : le système de suivi de bogues est un endroit trop encombré pour mener une discussion à laquelle, de toute façon, personne ne prêtera attention. On s'attend en effet à ce que le débat ait lieu sur la liste « développement » où une recherche peut se faire. Il est possible que certains ne soient même pas abonnés à la liste « résolution de problèmes » et, même s'ils le sont, ils ne suivent peut-être pas de près ce qu'il s'y passe.

Mais à quel moment précis du processus quelque chose est-il allé de travers ? Est-ce quand la personne du début a ajouté sa solution à l'endroit où le problème était décrit : aurait-elle dû plutôt la poster sur la liste ? où est-ce quand la deuxième personne a répondu au même endroit plutôt que sur la liste ?

Plutôt qu'une unique bonne réponse, voici un principe général : si vous ajoutez des données à un sujet faites-le sur le système de suivi, mais si vous démarrez une discussion faites-le sur la liste. Vous ne serez peut-être pas toujours en mesure de déterminer dans quelle catégorie ranger votre intervention, mais suivez votre jugement. Par exemple, au moment de joindre un correctif qui contient une solution qui peut prêter à controverse, vous devez anticiper le fait que les autres auront des questions à poser. Donc, même si vous auriez trouvé normal de joindre le correctif à la description du problème (en supposant que vous ne voulez ou ne pouvez pas valider le changement directement), vous devriez plutôt choisir de le poster sur la liste de diffusion. De toute façon, les échanges aboutiront à un point où l'une des parties dira que cela va plus loin que l'ajout pur et simple de données et qu'il faut une vraie discussion : dans l'exemple qui ouvre cette partie, ce serait à celui qui répond en deuxième, prenant conscience qu'il y a des problèmes sur le correctif, de prévoir qu'il en découlera une vraie discussion et qu'elle doit avoir lieu sur le support approprié.

Pour utiliser une analogie mathématique : si l'information vous semble pouvoir converger rapidement, mettez-la directement dans le système de suivi de bogues : si elle a l'air divergente, la liste de discussion ou le canal IRC semblent mieux appropriés.

Cela ne veut pas dire qu'il ne devrait jamais y avoir d'échanges dans le système de suivi. Demander des détails supplémentaires sur la façon de reproduire le bogue au rapporteur initial est un processus hautement convergent par exemple. La réponse de l'intéressé a peu de chances de soulever de nouvelles questions : il ne s'agit que d'étayer l'information référencée précédemment. Il n'y a pas lieu de perturber la liste avec ce processus. Essayez au maximum de régler cette question par une série de commentaires à même le système de suivi. De même, si vous êtes certain qu'un bogue a été rapporté à tort (c'est-à-dire qu'il ne s'agit pas d'un bogue), vous pouvez le dire directement sur le système de suivi. Même souligner un problème mineur à propos d'une solution proposée est acceptable, du moment que le problème en question ne compromet pas l'ensemble de la solution.

D'un autre côté, si vous soulevez des questions philosophiques sur la portée d'un bogue, ou sur le comportement correct du logiciel, vous pouvez avoir la certitude que d'autres développeurs voudront y participer. La discussion divergera vraisemblablement pendant un moment avant de converger de nouveau : menez-la donc sur la liste.

Donnez toujours des liens vers le système de suivi quand vous choisissez de poster sur la liste. Il est important que tous ceux qui suivent le bogue puissent accéder à la discussion même si l'endroit où il est rapporté n'est pas le lieu où se tient la discussion. La personne qui démarre le fil de discussion trouvera sans doute cela pénible, mais le milieu du logiciel libre est fondamentalement une culture de la responsabilité par l'écrit : il est bien plus important

de faciliter le travail aux dizaines ou centaines de personnes qui liront le bogue qu'aux trois ou quatre qui écrivent dessus.

On peut aussi extraire des conclusions ou des résumés importants de la liste de discussion pour les coller dans le système de suivi si cela aide les lecteurs. Un usage courant consiste à commencer une discussion dans la liste en mettant un lien vers le rapport de bogue, puis, une fois la discussion achevée, à coller le résumé dans le système de suivi (avec un lien vers le message qui contient le résumé, pour que ceux qui parcourent le suivi puissent voir facilement la conclusion à laquelle on est arrivé sans avoir à cliquer ailleurs). Notez que le problème courant des « deux originaux », la duplication des données, n'existe pas ici, car aussi bien les archives que les commentaires de bogues sont généralement des données statiques, non modifiables de toutes façons.

6. Les annonces

Dans le monde des logiciels libres, la frontière entre discussions internes et communiqués officiels est ténue. C'est en partie dû au fait que l'on ne sait jamais exactement à quel public on s'adresse : comme pratiquement tous les messages sont accessibles publiquement, le projet ne contrôle pas entièrement l'appréciation du public. Quelqu'un, disons un éditeur de Slashdot.org, pourrait attirer l'attention de millions de lecteurs sur un message que personne ne s'attendait à voir sortir du projet. C'est un aléa de la vie avec lequel tous les projets Open Source doivent composer, mais dans la pratique le risque est en général très faible. Habituellement les annonces que le projet veut rendre publiques sont celles qui seront le plus mises en avant, à condition que vous utilisiez les bons outils pour indiquer l'importance relative de votre communiqué au monde extérieur.

Pour les annonces majeures, on peut distinguer quatre à cinq canaux principaux de distribution au travers desquels les annonces devraient être faites aussi simultanément que faire se peut :

1. La page d'accueil de votre site Web est certainement la partie la plus visible du projet. Si vous devez faire une annonce particulièrement importante, affichez-y votre petit discours. Cela doit rester concis, un petit résumé qui renvoie vers le communiqué de presse (voir ci-dessous).
2. Parallèlement, vous devriez avoir aussi une section « Nouvelles » ou « Communiqués de presse » sur votre site Web où vous pourrez afficher toutes les annonces en détail. Les communiqués de presse servent de vitrine vers laquelle les autres sites peuvent rediriger leurs lecteurs. Assurez-vous donc qu'ils soient structurés soit sous la forme d'une page Web par communiqué, soit par un nouvel article distinct, ou sous n'importe quelle forme tant que l'on peut y accéder grâce à un lien sans qu'il y ait confusion possible avec d'autres communiqués.
3. Si vous avez créé un flux RSS pour votre projet, assurez-vous qu'il relaie également l'annonce. Cela peut se faire automatiquement lorsque vous créez le communiqué de presse selon la configuration de votre site Web (les flux RSS sont des outils pour distribuer des résumés de nouvelles contenant des métadonnées aux « abonnés »,

c'est-à-dire aux personnes qui ont fait en sorte de recevoir ces résumés. Voir l'article « What is RSS » de Mark Pilgrim [1] pour plus d'informations à propos des flux RSS).

4. Si l'annonce que vous passez concerne une nouvelle version, vous devez aussi modifier la page du projet sur http ://freshmeat.net (voir la section nommée « Annoncer » pour savoir comment créer cette page). Dès que vous modifiez votre page sur Freshmeat, la modification est annoncée sur la liste des changements du jour. Cette liste aussi est mise à jour sur d'autres portails (Slashdot.org compris) qui sont avidement surveillées par des hordes d'internautes. Freshmeat relaie également ces informations par son flux RSS. Ainsi, ceux qui ne sont pas abonnés à votre propre flux RSS pourront tout de même recevoir l'annonce par celui de Freshmeat.
5. Envoyez un e-mail à la liste de diffusion d'annonces de votre projet. Le nom de cette liste devrait d'ailleurs être « annonce », c'est-à-dire que l'adresse devrait être *annonce@domaineduprojet.org* parce que c'est devenu une norme maintenant. La charte de la liste devrait aussi annoncer clairement qu'elle engendrera l'envoi de peu de mails et qu'elle est réservée aux annonces du projet. La plupart des annonces concerneront les nouvelles versions du logiciel mais aussi, à l'occasion, d'autres évènements comme une collecte de fonds, la découverte d'une faille de sécurité (voir la section « Annoncer les failles de sécurité » plus tard dans ce chapitre), ou encore, un bouleversement dans la vie du projet pourront y être postés. Parce qu'elle génère peu de trafic, et qu'elle n'est employée que pour des choses importantes, la liste d'annonces possède en général le plus grand nombre d'abonnés parmi toutes les listes du projet (ce qui implique que vous ne devriez pas en abuser : tournez sept fois vos doigts au-dessus du clavier avant d'y poster). Pour éviter que n'importe qui y fasse des annonces, ou pire encore, qu'elle serve à envoyer des spams, la liste d'annonces doit toujours être modérée.

Il faut que les annonces soient faites aussi simultanément que possible sur tous ces outils. Les gens pourraient trouver bizarre de voir l'annonce faite sur la liste de diffusion et de ne pas la retrouver sur la page d'accueil du projet ou dans la section « Communiqués de presse ». Si vous pouvez préparer les différentes modifications (e-mails, modifications de pages Web, etc.) et les envoyer d'un coup, les unes à la suite des autres, la période de « contradiction » pourra être largement réduite.

Pour un évènement de moindre importance, vous pouvez réduire le nombre de canaux employés, voire n'en utiliser aucun. L'évènement sera tout de même remarqué par le monde extérieur à hauteur de son importance. Par exemple, alors que la sortie d'une nouvelle version d'un logiciel est un évènement majeur, le simple fait d'annoncer la date de la future version, même si cela reste une nouvelle importante, est loin d'être aussi capital que la sortie en elle-même. Quand une date est arrêtée pour la prochaine version, vous pouvez très bien vous contenter d'envoyer un mail sur les listes de diffusion générales (pas la liste d'annonces) et mettre à jour les prévisions du projet ou la page d'avancement.

Vous verrez malgré tout cette date apparaître dans les discussions sur d'autres sites Internet, partout où des gens sont intéressés par votre projet, qui suivent votre liste de diffusion, qui ne font que la consulter sans jamais y participer, et ne sont pas nécessairement muets par

1. http ://www.xml.com/pub/a/2002/12/18/dive-into-xml.html

ailleurs. Le bouche à oreille peut porter rapidement les nouvelles, vous devriez compter dessus et rédiger les annonces même mineures de manière à encourager la transmission exacte de l'information. En particulier, les messages que vous espérez voir cités devraient contenir une partie explicitement faite à cette fin, comme si vous écriviez un communiqué de presse officiel. Par exemple :

> *« Quelques nouvelles sur l'avancement : nous pensons sortir la version 2.0 de Scanley vers la mi-août 2005. Nous vous invitons à consulter la page http ://www.scanley.org/status.html régulièrement pour connaître les dernières nouvelles. La grosse nouveauté sera la recherche par expression rationnelle.*
>
> *Parmi les autres nouvelles fonctionnalités vous retrouverez :… Nous ajouterons également différentes corrections de bogues, à commencer par :… »*

Le premier paragraphe est succinct et donne les deux informations principales (la date de sortie et la grande nouveauté) ainsi que l'URL à visiter pour plus d'informations. Si ce paragraphe est le seul à être repris, vous aurez malgré tout réussi à passer l'information. Le reste du mail peut-être « oublié » sans amputer le message de son essence. Il y aura toujours des personnes qui mettront un lien vers le mail complet, mais il est aussi probable qu'ils n'en citeront qu'une partie. Puisque cette possibilité existe, autant leur simplifier la tâche et par la même occasion contrôler un peu mieux ce qui sera cité.

Annoncer les failles de sécurité

La gestion d'une faille de sécurité est différente de la gestion des autres rapports de bogue. Dans le logiciel libre, tout faire de manière ouverte et transparente relève presque du sacerdoce. Chaque étape d'une correction de bogue standard est visible aux yeux de tous ceux que cela intéresse : l'envoi du rapport initial, la discussion qui s'ensuit et le correctif.

Les bogues de sécurité sont différents. Ils peuvent mettre en danger les données des utilisateurs voire leur système entier. Parler de ce problème ouvertement alerterait tout le monde, y compris ceux qui voudraient en faire un usage malveillant. Le simple fait d'envoyer un correctif annonce l'existence du bogue (des agresseurs potentiels surveillent les journaux de commit des projets publics à la recherche de modifications qui indiquent des problèmes de sécurité). La plupart des projets Open Source se sont mis d'accord sur une série d'étapes à peu près standard pour gérer ce conflit entre ouverture et discrétion, elles sont basées sur quelques grandes lignes :

1. Ne pas parler du bogue en public tant qu'il n'y a pas de correctif disponible, fournir le correctif exactement au même moment où vous annoncez le bogue.
2. Concocter un correctif aussi rapidement que possible, surtout si c'est quelqu'un d'étranger au projet qui a rapporté le bogue, parce qu'alors vous savez qu'au moins une personne en dehors du projet est capable d'exploiter la vulnérabilité.

Dans la pratique, ces principes ont donné naissance à une série de mesures plus ou moins standardisées qui sont décrites dans les parties suivantes.

Réception du rapport

Pour commencer, le projet doit évidemment pouvoir recevoir un rapport de bogue de sécurité de n'importe qui. Mais l'adresse normale pour rapporter les bogues n'est pas adaptée ici parce qu'elle peut être vue par tout le monde. Il vous faudra donc une adresse différente pour recevoir les rapports de bogue de sécurité. Les archives de cette liste de diffusion ne doivent pas être visibles publiquement et les abonnés doivent être triés sur le volet, seuls les développeurs présents de longue date et sûrs peuvent être sur cette liste. S'il vous faut une définition plus concrète de « sûrs » voyez cela comme « toute personne qui possède l'accès de *commit* depuis deux ans au moins » ou quelque chose comme ça pour éviter le favoritisme. C'est le groupe qui devra gérer les failles de sécurité.

Idéalement cette liste de sécurité ne devrait pas être protégée du spam ou modérée sinon vous risqueriez d'évincer un rapport important ou de le retarder parce qu'aucun modérateur n'était disponible ce week-end. Si par contre vous utilisez un logiciel pour vous prémunir du spam utilisez des réglages tolérants, il vaut mieux laisser passer quelques spams que de manquer un rapport. Pour que cette liste soit efficace, vous devez évidemment rendre son adresse bien visible, mais comme elle ne sera pas modérée, et qu'au mieux elle sera faiblement protégée contre le spam, ne l'affichez jamais sans transformation, masquez-la comme nous l'avons vu dans la section « Masquer les adresses dans les archives » du chapitre 3. Heureusement, le masquage de l'adresse ne la rend pas nécessairement illisible, voir http ://subversion.tigris.org/security.html et jetez un œil à son code source pour y trouver un exemple.

Développer le correctif en secret

Que doit donc faire la liste de sécurité quand elle reçoit un rapport ? Elle doit en premier lieu évaluer la sévérité du problème et son urgence :

1. Quelle est la gravité de la vulnérabilité ? Est-ce qu'elle permet à un agresseur malintentionné de prendre le contrôle de l'ordinateur d'une personne utilisant le logiciel ? Ou ne fait-elle, par exemple, que divulguer des informations à propos de la taille de certains fichiers ?
2. Avec quelle facilité peut-on exploiter cette vulnérabilité ? Une attaque peut-elle être scriptée ou requiert-elle une connaissance détaillée, du raisonnement et de la chance ?
3. Qui a rapporté le problème ? La réponse à cette question ne change pas la nature de la vulnérabilité évidemment, mais cela vous donne une idée du nombre de personnes qui pourraient être au courant. Si le rapport provient de l'un des développeurs du projet, vous pouvez vous détendre un peu (mais seulement un peu), en effet, vous pouvez lui faire confiance pour ne pas l'avoir ébruité. À l'opposé, si c'est un e-mail de *anonyme14@globalhackers.net* que vous recevez, vous devriez agir au plus vite. Cette personne vous a rendu service en vous informant du problème, mais vous n'avez aucune idée du nombre de personnes qu'elle a pu mettre au courant ou de combien de temps elle attendra avant d'exploiter la faille à grande échelle.

Comme vous l'avez remarqué, l'éventail d'urgence ne s'étend que de « urgent » à « extrêmement urgent ». Même si le rapport provient d'une source connue et inoffensive, il peut très bien y avoir d'autres personnes sur le Net qui ont découvert le bogue depuis longtemps mais qui ne l'ont pas rapporté. Le seul cas de figure où il n'y a pas vraiment urgence est quand par sa nature le bogue ne pose pas de risque important de sécurité.

Et au fait, l'exemple de *anonyme14@globalhackers.net* n'est pas facétieux. Il est fort probable que vous receviez des rapports de bogues de personnes masquant leur identité et qui, par leurs mots ou leur comportement, n'établissent jamais clairement s'ils sont de votre côté ou contre vous. Cela n'a pas d'importance : s'ils vous ont rapporté la faille de sécurité, ils se diront qu'ils vous ont fait une faveur et vous devriez répondre en conséquence. Remerciez-les pour le rapport, donnez leur un délai dans lequel vous prévoyez de fournir un correctif public, et gardez-les dans la confidence. Parfois ils vous donneront une date, c'est une menace implicite de publier le bogue à une échéance donnée, que vous soyez prêt ou non. Ça peut ressembler à de l'intimidation, mais il faut plutôt y voir une action préventive, fruit de déceptions passées dues au manque de réactivité de fabricants de logiciels qui n'ont pas pris ces rapports de sécurité au sérieux. Quoi qu'il en soit, vous ne pouvez pas vous permettre d'ignorer cette personne. Après tout, si le bogue est important, elle dispose des connaissances nécessaires pour vous causer de gros ennuis. Traitez ces rapporteurs correctement en espérant qu'ils en feront de même en retour.

Un autre cas fréquent est le rapport de bogue rédigé par un professionnel en sécurité, quelqu'un qui gagne sa vie en inspectant les codes et qui garde toujours un œil sur les dernières vulnérabilités des logiciels. Ces personnes possèdent en général la double expérience d'avoir déjà reçu des rapports de bogues et d'en avoir envoyé aussi, sûrement plus que la plupart des développeurs de votre projet. Ils sont aussi coutumiers des dates limites imposées pour réparer le problème avant de dévoiler la faille au public. Dans certains cas vous pouvez négocier cette date, mais tout dépend du rapporteur. Le fait de fixer une date limite s'est imposé petit à petit dans le monde des professionnels en sécurité comme le seul moyen sûr pour que les entreprises répondent rapidement aux problèmes de sécurité. Ne voyez donc pas ces dates limites comme une pratique grossière : si cette manière de faire s'est imposée avec le temps, c'est qu'il y a de bonnes raisons.

Une fois la sévérité et l'urgence établies, vous pouvez commencer à travailler sur le correctif. Il faut trouver le bon équilibre entre faire les choses avec élégance et faire les choses rapidement. C'est la raison pour laquelle il faut déterminer l'urgence de la situation avant de commencer. Il faut que la discussion concernant le correctif reste entre les membres de la liste de sécurité et le rapporteur initial (s'il veut être impliqué) et tout développeur dont les compétences seront nécessaires.

N'enregistrez pas le correctif dans le dépôt. Il faut le garder à l'écart jusqu'à la date de publication. Si vous l'enregistriez, même avec un message de journal innocent, quelqu'un pourrait le remarquer et comprendre la modification. Vous ne savez jamais qui surveille votre dépôt ou pour quelles raisons. Arrêter les e-mails de commit n'arrangerait rien, en premier lieu parce qu'un trou dans la suite des courriers serait suspect et de toutes façons les données se retrouveraient dans le dépôt. Contentez-vous de mener tout le développement hors du dépôt, et conservez ce travail dans un endroit secret, pourquoi pas un dépôt privé, distinct, connu des seules personnes déjà au courant du bogue (si vous utilisez un logiciel de

gestion de versions décentralisé comme Arch ou SVK vous pouvez travailler sous gestion de versions et simplement empêcher l'accès à ce dépôt aux personnes externes).

Les numéros CAN/CVE

Vous avez déjà peut-être rencontré un numéro CAN ou CVE associé à un problème de sécurité. Il se présente en général sous cette forme : « CAN-2004-0397 » ou « CVE-2002-00923 ».

Ces deux types de numéros représentent la même chose : une entrée dans la liste des « Common Vulnerabilities and Exposures » ou « Vulnérabilités et Failles Courantes », cette liste est disponible à l'adresse http ://cve.mitre.org/. Son but est de fournir des noms normalisés à tous les problèmes de sécurité connus afin que chacun ait un nom canonique unique que l'on peut employer pour le désigner et de fournir un site de centralisation où l'on peut se rendre pour obtenir plus d'informations. La seule différence entre les numéros « CAN » et « CVE » est que le premier représente une demande d'inclusion (*candidate entry*), pas encore approuvée pour l'ajout à la liste officielle par l'équipe rédactionnelle de CVE, et que le dernier désigne une entrée approuvée. Ces deux types d'entrées sont de toute façon visibles par le public et le numéro de l'entrée n'est pas modifié lorsqu'elle est approuvée, le préfixe « CAN » est simplement remplacé par « CVE ».

Une entrée CAN/CVE ne renferme pas une description complète du bogue ou de la manière de s'en prémunir. On peut y trouver un bref résumé et une liste de liens vers des ressources externes (comme par exemple des archives de listes de diffusion) où peuvent se rendre les gens pour obtenir des informations plus détaillées. La vraie utilité de http ://cve.mitre.org/ est de fournir un espace bien organisé dans lequel chaque vulnérabilité peut avoir un nom et où l'on peut trouver des liens vers des données plus complètes. Visitez la page CVE-2002-0092 [1], vous y trouverez un exemple d'entrée. Attention, les références peuvent être très laconiques et les sources peuvent apparaître sous forme d'abréviations mystérieuses. Un glossaire [2] des abréviations est cependant disponible.

Si votre vulnérabilité remplit les critères de CVE, vous pouvez postuler pour un numéro CAN. Le processus pour cela est délibérément fermé, vous devez connaître quelqu'un, ou quelqu'un qui connaît quelqu'un. Ce n'est pas aussi étrange que cela puisse paraître. Afin d'éviter que l'équipe rédactionnelle de CVE ne croule sous les demandes mal rédigées, ne sont acceptées que les soumissions issues de sources de confiance. Si vous souhaitez voir votre vulnérabilité listée, vous devez donc d'abord trouver un intermédiaire entre votre projet et l'équipe rédactionnelle de CVE. Interrogez d'abord vos développeurs, il se peut très bien que l'un d'entre eux connaisse déjà quelqu'un qui serait passé par le processus CAN avant, ou qu'il connaisse quelqu'un d'autre encore qui l'aurait déjà fait, etc. L'avantage de cette manière de procéder est que, quelque part sur l'échelle de connaissances, quelqu'un peut être suffisamment renseigné pour vous dire que a) votre demande ne répond pas aux critères de MITRE et donc que ce n' est pas la peine de la faire ou que b) cette vulnérabilité possède déjà un numéro CAN ou CVE. Ce dernier cas peut se produire si le bogue a déjà été publié sur une autre liste de sécurité, par exemple à l'adresse cert.org ou sur la liste de diffusion de

1. http ://cve.mitre.org/cgi-bin/cvename.cgi ?name=2002-0092
2. http ://cve.mitre.org/cve/refs/refkey.html

BugTraq [1] (si cela se produit sans que votre projet en entende parler alors vous devriez vous demander quels autres événements vous avez loupés).

Si vous réussissez finalement à obtenir un numéro CAN/CVE il vaut mieux l'avoir tout au début de vos recherches concernant le bogue afin que toutes les communications ultérieures puissent se référer à ce numéro. Il y a un embargo maintenu sur les entrées CAN jusqu'à leur date de publication, l'entrée reste vide mais sert à réserver le numéro afin de ne pas le perdre, mais elle ne révélera aucune information à propos de la vulnérabilité jusqu'à la date à laquelle vous annoncerez le bogue et le correctif.

Vous trouverez plus d'informations [2] à propos du processus CAN/CVE sur le site cve.mitre.org ainsi qu'une explication [3] particulièrement claire de l'utilisation de la numérotation CAN/CVE par un projet Open Source sur le site Debian.org.

Pré-notification

Une fois que votre équipe de sécurité (c'est-à-dire les développeurs de la liste de diffusion de sécurité plus les personnes à qui on a fait appel pour leurs compétences) a préparé un correctif, vous devez décider de la manière de le distribuer.

Si vous ajoutez simplement ce correctif à votre dépôt, ou si vous mettez le monde au courant d'une quelconque manière, vous faites peser la menace d'une attaque sur vos utilisateurs qui seront alors obligés de mettre à jour le logiciel. Il est donc parfois de bonne pratique de pré-notifier certains utilisateurs importants. C'est particulièrement vrai pour les logiciels client/serveur utilisables par des serveurs bien connus qui pourraient devenir des cibles privilégiées. Les administrateurs de ces serveurs vous seraient reconnaissants d'avoir un délai d'un jour ou deux pour faire la mise à jour afin d'être déjà protégés lorsque la faille est rendue publique.

Vous avez simplement besoin d'envoyer un mail à ces administrateurs avant la date de publication les informant de la vulnérabilité et des solutions. Vous ne devriez envoyer ces pré-notifications qu'aux personnes de confiance. Pour rentrer dans cette catégorie, il existe deux conditions : le destinataire doit administrer un serveur important où un risque pourrait avoir des conséquences graves, et le destinataire doit avoir la réputation de quelqu'un qui n'ira pas ébruiter le problème de sécurité avant la date de publication.

Envoyez chaque e-mail de pré-notification individuellement (un par un) à chaque destinataire. Ne l'envoyez pas à une liste entière de destinataires en une fois parce qu'alors chacun verrait le nom des autres et ce serait comme les avertir que tous les autres destinataires ont peut-être une faille de sécurité sur leur serveur. Il ne faut pas non plus les envoyer en copie cachée (Bcc :) parce que les administrateurs protègent leurs boîtes de réception avec des filtres anti-spam qui bloquent ou abaissent la priorité des courriers reçus en Bcc :.

Voici un exemple de mail de pré-notification :

1. http ://www.securityfocus.com/
2. http ://cve.mitre.org/about/candidates.html
3. http ://www.debian.org/security/cve-compatibility

> De : Votre nom À : admin@gros-serveur.com Répondre à : Votre nom (pas à la liste de sécurité) Sujet : Notification de vulnérabilité confidentielle dans Scanley
>
> Cet e-mail est confidentiel, c'est une pré-notification pour une alerte de sécurité affectant le serveur Scanley.
>
> Merci de *ne transmettre ce message à personne*. L'annonce publique est prévue pour le 19 mai et nous aimerions que l'information reste confidentielle jusqu'à cette date.
>
> Cet e-mail vous est adressé car (nous croyons que) vous utilisez un serveur Scanley et vous désirez sûrement appliquer le correctif avant que la faille ne soit rendue publique le 19 mai.
>
> Référence : ========== CAN-2004-1771 : Scanley stack overflow in queries
>
> Vulnérabilité : ========== Il est possible de faire exécuter des commandes au hasard au serveur si le serveur est mal configuré et que le client envoie une requête mal conçue.
>
> Contournement : ========== Désactiver l'option 'natural-language-processing' dans scanley.conf ferme la faille.
>
> Correctif : ========== Le correctif qui suit s'applique à Scanley 3.0, 3.1 et 3.2.
>
> Une nouvelle version publique (Scanley 3.2.1) sera publiée le 19 mai ou juste avant afin d'être disponible au moment où la vulnérabilité sera divulguée. Vous pouvez appliquer le correctif maintenant ou simplement attendre la sortie publique. La seule différence entre les versions 3.2 et 3.2.1 sera ce correctif.
>
> [Insérer le correctif ici...]

Si vous avez un numéro CAN, indiquez-le dans la pré-notification (comme ci-dessus), même si l'information est toujours sous embargo et que la page MITRE est donc encore vide. En ajoutant le numéro CAN vous êtes sûr que le destinataire sait que le bogue dont vous lui envoyez la pré-notification est le même que celui dont il entendra parler peu après par voie publique. Ainsi il n'a pas à se demander s'il doit prendre d'autres mesures ou non, ce qui est précisément le but des numéros CAN/CVE.

Distribuez le correctif publiquement

La dernière chose qu'il vous reste à faire est de distribuer le correctif publiquement. Dans une annonce unique et complète, vous devez décrire le problème, donner le numéro CAN/CVE s'il y en a un, décrire comment contourner le bogue et comment le régler définitivement. « Régler définitivement » implique souvent la mise à jour du logiciel, même si

parfois cela peut simplement vouloir dire appliquer un correctif, surtout si le logiciel est normalement utilisé sous forme de source. Si vous proposez une nouvelle version du logiciel, celle-ci ne doit différer de la précédente que par le correctif de sécurité. Les administrateurs plutôt conservateurs pourront ainsi faire la mise à jour sans se préoccuper d'autres conséquences. Ils n'ont pas à se soucier non plus des mises à jour futures puisqu'ils sauront que le correctif y sera également inclus (les détails concernant les procédures de publication sont abordés dans la section nommée « Mises à jour de sécurité » dans le chapitre 7).

Que le correctif public implique une nouvelle version ou non, faites l'annonce avec plus ou moins la même priorité que pour une nouvelle version : envoyez un mail à la liste de diffusion d'annonces, écrivez un nouveau communiqué de presse, mettez à jour votre entrée sur Freshmeat, etc. Vous ne devriez jamais essayer de minimiser l'existence d'une faille de sécurité si vous vous souciez un tant soit peu de la réputation de votre projet. Mais gardez toujours à l'esprit qu'il faut adapter le ton et la visibilité d'une annonce de sécurité à la sévérité concrète du problème. Si le bogue de sécurité n'est qu'un risque mineur de révélation d'informations, pas une faille qui permettrait la prise de contrôle totale de l'ordinateur de l'utilisateur, alors il ne devrait pas faire tant de bruit. Vous n'aurez peut-être même pas besoin de déranger la liste d'annonces pour ça. Après tout, si le projet crie au loup à chaque fois, les utilisateurs finiront par croire que le logiciel est moins sécurisé qu'il ne l'est vraiment et pourraient également ne pas vous croire si vous aviez un vrai problème à annoncer. Voir http ://cve.mitre.org/about/terminology.html, cette page constitue une bonne introduction à l'évaluation de la sévérité d'un bogue.

De manière générale, si vous n'êtes pas certain des suites à donner à un problème de sécurité, trouvez quelqu'un d'expérience avec qui vous pourrez en discuter. Évaluer et gérer les vulnérabilités, cela ne s'apprend pas du jour au lendemain, il est courant de faire des erreurs les premières fois.

Paquets, sorties et développement quotidien

Ce chapitre se rapporte à la création et à la publication de logiciels par les projets de logiciels libres ainsi qu'à l'organisation du processus de développement tout entier autour de ces objectifs.

L'une des grandes différences entre les projets Open Source et les projets propriétaires est l'absence de contrôle centralisé sur l'équipe de développement. C'est lorsqu'arrive le temps de préparer une nouvelle sortie que cette différence est particulièrement frappante : une entreprise peut demander à toute son équipe de développement de se concentrer sur la sortie en préparation en mettant de côté le développement de nouvelles fonctionnalités et les corrections de bogues mineurs jusqu'à la publication de la nouvelle version. Les groupes de volontaires ne sont pas aussi monolithiques. Les membres travaillent sur le projet pour de nombreuses raisons différentes, et sont libres de ne pas chambouler leurs priorités à cause d'une sortie imminente. Comme le développement ne s'arrête jamais, les procédures de sortie dans un environnement Open Source tendent à être plus longues, mais sont moins perturbantes que celles des logiciels commerciaux. On peut faire une analogie avec l'entretien d'une autoroute. Vous avez deux options pour la réparer : vous pouvez la fermer complètement afin que les équipes de travaux puissent l'envahir à loisir jusqu'à ce que le problème soit résolu, ou bien vous pouvez travailler sur quelques voies en laissant les autres ouvertes au trafic. La première solution est très efficace pour l'équipe de réparation, mais la voie est complètement fermée aux autres jusqu'à la fin du chantier. La deuxième solution implique beaucoup plus de contraintes pour l'équipe de chantier (ils doivent travailler avec moins de monde et moins d'équipement, dans des conditions plus restreintes, affecter des préposés à la circulation, etc.), mais au moins la voie reste utilisable, même si ce n'est pas à sa capacité maximale.

Les projets Open Source ont tendance à adopter la deuxième solution. En fait, pour un logiciel évolué ayant plusieurs lignes de sortie maintenues simultanément, le projet est en

quelque sorte toujours en travaux. Il y a toujours quelques voies fermées, un niveau constant, mais bas, d'incommodité est toujours toléré par l'ensemble du groupe afin que les sorties puissent être faites à intervalles réguliers.

Le modèle qui rend tout cela possible ne s'applique pas qu'aux sorties. C'est le principe de parallélisation des tâches qui ne sont pas mutuellement interdépendantes, un principe qui n'est en aucun cas spécifique au développement Open Source évidemment, mais un principe que les projets Open Source adaptent à leurs besoins particuliers. Ils ne peuvent pas se permettre de trop importuner l'équipe de chantier, ou le trafic habituel, mais ils ne peuvent pas non plus se permettre d'employer du personnel spécialement pour se poster près des plots oranges et agiter un drapeau pour faire la circulation. Ils s'orientent donc vers des processus qui demandent un niveau soutenu d'administration plutôt que vers des processus irréguliers. Les volontaires s'accommodent très bien d'un niveau constant, mais faible, de désagréments. Comme tout est prévisible, ils peuvent aller et venir sans se demander si leur emploi du temps entrera en conflit avec ce qui se passe dans le projet. Mais si le projet était soumis à un emploi du temps dans lequel certaines activités en excluraient d'autres, on se retrouverait finalement avec beaucoup de développeurs n'ayant rien à faire pendant une grande partie de ce temps, ce qui serait non seulement inefficace, mais aussi ennuyeux, et donc dangereux (il y a de fortes chances qu'un développeur qui s'ennuie quitte le projet).

Le travail de publication est généralement celui qui est le plus visible parmi les tâches qui ne sont pas directement liées, mais parallèles au développement. Ainsi, les méthodes décrites dans les parties suivantes concernent principalement les sorties. Cependant, notez qu'elles s'appliquent également à d'autres tâches parallélisables, comme les traductions et l'internationalisation, les vastes changements d'API apportés graduellement à l'ensemble du code source, etc.

1. Numérotation des versions

Avant d'aborder la conception d'une version, parlons d'abord de la façon de nommer les versions, et pour cela, il faut que vous connaissiez la signification des sorties pour les utilisateurs. Une sortie signifie que :

- Les anciens bogues ont été corrigés. C'est quasiment un impératif à chaque sortie.
- De nouveaux bogues ont été ajoutés. Les utilisateurs s'y attendent aussi, mais pas quand il s'agit d'un correctif de sécurité ou de quelques autres cas exceptionnels (voir la section appelée « Correctifs de sécurité » plus loin dans ce chapitre).
- De nouvelles fonctionnalités peuvent avoir été ajoutées.
- De nouvelles options de configuration peuvent avoir été ajoutées et certaines anciennes options ont pu légèrement changer. Les procédures d'installation peuvent, également, avoir été légèrement modifiées depuis la version précédente également, bien qu'en général ce ne soit pas souhaitable.
- Des changements apportés peuvent rendre certaines données incompatibles et le format de données utilisé dans d'anciennes versions du logiciel n'est plus utilisable sans passer par une étape de conversion unidirectionnelle (potentiellement manuelle).

Comme vous pouvez le voir, ce ne sont pas que de bonnes choses. C'est la raison pour laquelle les utilisateurs expérimentés appréhendent toujours un peu les sorties, surtout quand

le logiciel est mature et réalise déjà presque tout ce qu'ils veulent faire avec (ou du moins, espèrent pouvoir faire). Même l'arrivée de nouvelles fonctionnalités a ses avantages et ses inconvénients puisque le comportement du logiciel peut être modifié.

La numérotation a donc deux buts : de toute évidence le numéro devrait indiquer sans ambiguïté l'ordre de sortie (c'est-à-dire qu'en regardant deux numéros de version, on devrait savoir quelle est la plus récente), mais elle devrait également indiquer de manière aussi condensée que possible l'importance et la nature des changements de cette version.

Tout cela dans un numéro ? Eh bien, plus ou moins, oui. La stratégie de numérotation des sorties est une des plus vieilles controverses (voir la section appelée « Plus le sujet est facile, plus long sera le débat » dans le chapitre 6) et je doute que l'on puisse s'entendre sur une norme unique et complète d'ici peu. Cependant, quelques bonnes stratégies se sont démarquées et un principe est accepté par tous : soyez cohérents. Choisissez une stratégie de numérotation, décrivez-la et restez-y fidèle. Les utilisateurs vous en seront reconnaissants.

Les éléments de la numérotation

Cette partie décrit en détail les conventions formelles de la numérotation de version et ne nécessite que peu de connaissances préalables. Vous devez la considérer comme une référence. Si vous êtes déjà familier avec ces conventions, vous pouvez vous rendre directement à la section suivante.

Les numéros de versions sont des groupes de chiffres séparés par des points :

Scanley 2.3

Singer 5.11.4

… et ainsi de suite. Les points ne sont pas des virgules pour séparer les décimales, ce sont simplement des séparateurs ; *5.3.9* sera suivi par *5.3.10*. Certains projets ont parfois dévié de cette voie, l'exemple le plus connu est le noyau Linux avec sa suite *0.95*, *0.96*, *0.99* pour arriver à *Linux 1.0*, mais la convention, selon laquelle les points ne séparent pas les décimales, est maintenant bien établie et devrait être considérée comme une norme. Il n'y a aucune limite au nombre de composantes (portions de chiffres ne contenant pas de points), mais la plupart des projets ne dépassent pas trois ou quatre. Les raisons vous apparaîtront plus clairement par la suite.

En plus de la composante chiffrée, les projets ajoutent encore une étiquette telle que *Alpha* ou *Beta*, par exemple :

Scanley 2.3.0 (Alpha)

Singer 5.11.4 (Beta)

Une étiquette *Alpha* ou *Beta* signifie que cette sortie précède une sortie prochaine avec le même numéro, mais sans l'étiquette. Ainsi, *2.3.0 (Alpha)* aboutira finalement à *2.3.0*. Afin de laisser de la place pour plusieurs pré-sorties de suite, les étiquettes elles-mêmes peuvent porter une méta-étiquette. Par exemple, voici une série de sorties dans l'ordre où elles seraient mises à disposition du public :

Scanley 2.3.0 (Alpha 1)
Scanley 2.3.0 (Alpha 2)

Scanley 2.3.0 (Beta 1)
Scanley 2.3.0 (Beta 2)
Scanley 2.3.0 (Beta 3)
Scanley 2.3.0

Vous remarquerez que, quand il possède le qualificatif *Alpha*, *Scanley 2.3* est écrit comme *2.3.0*. Les deux nombres sont équivalents, le *0* en fin de numérotation peut toujours être abandonné par souci de concision, mais lorsqu'un qualificatif est présent, on utilise plutôt le nom complet.

D'autres qualificatifs utilisés relativement souvent sont *Stable*, *Unstable*, *Development* et *RC* (pour « Release Candidate »). Les plus utilisés restent *Alpha* et *Beta*, *RC* n'est pas loin derrière à la troisième place, mais vous remarquerez que *RC* est toujours suivi d'un méta-qualificatif numérique. C'est-à-dire que vous ne publiez pas *Scanley 2.3.0 (RC)*, vous publiez *Scanley 2.3.0 (RC1)*, suivi par *RC2*, etc.

Ces trois étiquettes, *Alpha*, *Beta* et *RC* sont largement connues maintenant, car ce sont des mots compréhensibles, pas du jargon, et je ne recommanderai pas l'usage des autres, même si *a priori* ce sont de meilleurs choix. Les gens qui installent des logiciels non-finalisés sont déjà familiers avec ce triptyque, et il n'y a pas de raison de réinventer la roue.

Même si les points dans les numéros de version ne sont pas des virgules, ils indiquent tout de même un certain ordre. Toutes les versions *0.X.Y* précèdent la *1.0* (qui est équivalente à la *1.0.0*, bien sûr). La *3.14.158* précède immédiatement la *3.14.159* et ne précède pas directement la *3.14.160* ou encore la *3.15.0* et ainsi de suite.

Une politique cohérente de numérotation de version permet aux utilisateurs de distinguer, uniquement en comparant les chiffres pour un même logiciel, les différences importantes qui existent entre deux variantes. Dans le cas classique d'une numérotation à trois composantes, le premier est le nombre principal, le deuxième est le nombre mineur et le troisième est le micro-nombre. Par exemple, la version *2.10.17* est la dix-septième micro sortie de la dixième sortie mineure au sein de la deuxième version principale. Les mots « lignes » et « séries » sont utilisés de manière informelle ici, mais ils veulent dire ce qu'ils veulent dire. Une série majeure comprend toutes les sorties qui partagent le même nombre principal, et une série mineure (une ligne mineure) contient toutes les versions qui partagent le même nombre mineur et le même nombre principal. Par exemple, la *2.4.0* et la *3.4.1* ne sont pas dans la même série mineure, même si elles ont toutes les deux le *4* comme chiffre mineur, à l'opposé, la *2.4.0* et la *2.4.2* sont dans la même ligne mineure bien qu'elles ne soient pas adjacentes si la *2.4.1* a été publiée entre les deux.

La signification de ces nombres est très logique et sans surprise : une incrémentation du nombre principal indique que de grands changements se sont produits, une incrémentation du nombre mineur indique des modifications mineures et une incrémentation du micro-nombre indique des changements vraiment triviaux. Certains projets ajoutent une quatrième composante, appelée habituellement le numéro du correctif, pour un contrôle particulièrement fin des différences entre les versions (certains projets utilisent « correctif » comme synonyme de « micro » dans un système à trois composantes, ce qui peut prêter à confusion). Il y a aussi d'autres projets qui utilisent la dernière composante comme un numéro de

« build ». Tous les rapports de bogue sont ainsi liés à un « build » particulier. Cette méthode est sûrement la mieux adaptée pour les logiciels distribués sous forme de paquets binaires.

Bien qu'il existe de nombreuses conventions différentes concernant le nombre de composantes que vous devriez utiliser et leur signification, les différences restent minimes, vous avez une certaine marge, mais pas tant que ça. Les deux prochaines parties portent sur les conventions les plus utilisées.

La stratégie simple

Chaque projet possède ses propres règles définissant les modifications autorisées dans une version si l'on incrémente seulement le micro-nombre, d'autres règles pour le nombre mineur et d'autres encore pour le nombre principal. S'il n'existe aucune norme encore, voici celle qui est appliquée avec succès par plusieurs projets. Même si votre projet n'adopte pas cette numérotation, vous trouverez dans les paragraphes suivants un bon exemple du type d'informations que doit véhiculer le numéro de version. Ces règles sont tirées du système de numérotation utilisé par le projet APR, voir http ://apr.apache.org/versioning.html.

1. Les modifications du micro-nombre seul (c'est-à-dire les modifications au sein de la même ligne mineure) doivent être compatibles à la fois avec les versions suivantes et les versions précédentes. Les modifications ne devraient être, par conséquent, que des correctifs de bogue ou de petites améliorations de fonctionnalités déjà existantes. De nouvelles fonctionnalités ne devraient pas être introduites dans une micro-sortie.
2. Les modifications du nombre mineur (c'est-à-dire, dans la même ligne principale) doivent être rétro-compatibles, mais pas nécessairement avec les versions futures. Il est normal d'introduire de nouvelles fonctionnalités dans une sortie mineure, mais en nombre limité normalement.
3. Des modifications du nombre majeur marquent des limites de compatibilité. Une nouvelle sortie majeure peut être compatible avec les versions suivantes et précédentes. Une sortie majeure est censée posséder de nouvelles fonctionnalités, et peut même avoir tout un nouvel ensemble de fonctionnalités.

La signification exacte de la compatibilité avec les versions précédentes et suivantes dépend de ce que le logiciel fait, mais dans le contexte du logiciel la signification est relativement claire. Par exemple, si votre projet est une application client/serveur, alors « rétro-compatible » signifie que mettre à jour le serveur avec la version *2.6.0* ne devrait pas empêcher les clients utilisant la version *2.5.4* de fonctionner comme avant (hormis pour les bogues corrigés évidemment). D'un autre côté, mettre à jour l'un de ces clients avec la version *2.6.0*, en même temps que le serveur, peut apporter de nouvelles fonctionnalités au client, des fonctionnalités que les clients avec la version *2.5.4* ne peuvent pas utiliser. Si cela se produit, alors la mise à jour n'est pas compatible : vous ne pouvez évidemment pas revenir en arrière et remettre la version *2.5.4* sur le poste client en gardant les fonctionnalités obtenues avec la *2.6.0* puisque certaines de ces fonctionnalités sont apparues avec la version *2.6.0*.

C'est pourquoi les micro-sorties ne font quasiment que corriger des bogues. Elles doivent être compatibles dans les deux sens : si vous faites une mise à jour de la version *2.5.3* vers

la version *2.5.4*, et qu'ensuite vous changez d'avis pour revenir à la version *2.5.3*, vous ne devriez pas perdre de fonctionnalités. Évidemment, les bogues corrigés dans la version *2.5.4* ré-apparaîtront après le retour en arrière, mais vous ne devez pas perdre de fonctionnalité, sauf si, éventuellement, un bogue corrigé par la nouvelle version empêche l'exécution de fonctionnalités existantes.

Les protocoles client/serveur ne représentent qu'un domaine de compatibilité parmi tant d'autres. On peut également citer le format des données : est-ce que le logiciel écrit des données sur une unité de stockage permanent ? Si c'est le cas, les formats de lecture et d'écriture doivent répondre aux mêmes normes imposées par les règles de numérotation. La version *2.6.0* doit pouvoir lire les fichiers écrits avec la version *2.5.4* mais peut très bien mettre à jour discrètement le format, le rendant non-interopérable avec la version *2.5.4*. La compatibilité avec les versions antérieures n'est pas requise quand on change de numéro mineur. Si votre projet distribue des librairies de code que d'autres programmes emploient, alors les API sont aussi un autre domaine de compatibilité : vous devez vous assurer que les règles de compatibilité de la source et des binaires sont bien énoncées de manière à ce que les utilisateurs informés n'aient pas à se demander s'ils peuvent mettre à jour leur version sans risque. Ils pourront regarder la numérotation et seront instantanément fixés.

Ce système vous empêche de changer les choses profondément tant que vous n'incrémentez pas le nombre majeur. Cela peut souvent devenir un vrai handicap : vous avez peut-être envie d'ajouter des fonctionnalités, ou de repenser certains protocoles, mais vous ne pourrez pas le faire par souci de compatibilité. Il n'existe pas de solution miracle, à part peut-être si, dès le départ, vous décidez de tout construire de façon extensible (il faudrait un livre entier pour couvrir ce sujet, il dépasse largement les buts de cet ouvrage). Mais vous devez publier des règles de compatibilité et vous y tenir, c'est une obligation. Une mauvaise surprise pourrait vous mettre de nombreux utilisateurs à dos. Les règles décrites précédemment sont bonnes, en partie parce qu'elles sont répandues, mais aussi parce qu'elles sont simples à expliquer et à mémoriser, même pour ceux qui n'en sont pas familiers.

De manière générale, ces règles ne s'appliquent pas aux versions précédant la version *1.0* (bien que vos règles puissent l'établir explicitement, juste pour être clair). Un projet qui en est toujours à un stade de développement initial peut sortir les versions *0.1*, *0.2*, *0.3* et ainsi de suite l'une après l'autre jusqu'à ce qu'il soit prêt pour la version *1.0*, et les différences entre ces versions peuvent être arbitrairement importantes. Les micro-nombres pour les sorties avant la *1.0* sont optionnels. Selon la nature de votre projet et les différences entre les sorties, vous jugerez utile ou non d'avoir des versions *0.1.0*, *0.1.1*, etc. Les conventions pour les sorties précédant la version *1.0* ne sont pas très strictes, les gens comprennent que de fortes contraintes de compatibilité entraveraient le développement initial et les premiers utilisateurs pardonnent facilement de toute manière.

Souvenez-vous que toutes ces règles ne s'appliquent qu'à ce système, à trois composantes particulier. Votre projet pourrait facilement élaborer un système différent à trois composantes, ou même décider que vous n'avez pas besoin d'une telle finesse, et plutôt adopter un système à deux composantes. Il faut surtout que vous décidiez rapidement du système que vous allez adopter, que vous publiiez ce que signifient exactement les composantes et vous y tenir.

La stratégie pair/impair

Certains projets se servent de la parité du nombre mineur pour indiquer la stabilité du logiciel : pair signifie stable, impair signifie instable. Ceci ne s'applique qu'au nombre mineur, pas au nombre principal ou aux micro-nombres. L'incrémentation du micro-nombre indique toujours des corrections de bogues (pas de nouvelles fonctionnalités) et l'incrémentation du nombre principal indique toujours de grands changements, un ensemble de nouvelles fonctionnalités, etc.

L'avantage du système pair/impair, qui est employé par le projet du noyau Linux entre autres, est qu'il offre un moyen de sortir de nouvelles fonctionnalités, pour qu'elles soient testées, sans exposer les utilisateurs à un code potentiellement instable. Les gens comprennent grâce aux chiffres que la version *2.4.21* est suffisamment stable pour être installée sur leur serveur Web, mais que la version *2.5.1* devrait rester confinée aux expérimentations sur une machine de test. L'équipe de développement traite les rapports de bogue qui arrivent pour la série instable (dont le nombre mineur est impair), et quand les choses commencent à se calmer, après quelques micro-sorties dans cette série, ils incrémentent d'une unité le nombre mineur, le rendant donc pair, remettent le micro-nombre à *0* et sortent un paquet présumé stable.

Ce système préserve les indications de compatibilité données précédemment, ou du moins n'entre pas en conflit avec. Le nombre mineur véhicule simplement plus d'informations. Son incrémentation est deux fois plus fréquente que nécessaire, mais il n'y a rien de grave là-dedans. Le système pair/impair est sans doute mieux adapté aux projets qui ont de longs cycles de développement et qui, par nature, ont une grande proportion d'utilisateurs conservateurs qui préfèrent la stabilité à de nouvelles fonctionnalités. Cependant, ce n'est pas l'unique moyen pour que ces nouvelles fonctionnalités soient testées en « situation réelle ». La partie de ce chapitre intitulée « Stabiliser une version » en décrit une autre, peut-être plus commune, pour proposer du code potentiellement instable au public, avec un avertissement dans le nom grâce auquel on peut évaluer plus simplement le rapport risque/bénéfice.

2. Les branches de publication

Du point de vue d'un développeur Open Source, le projet se trouve dans un état permanent de finalisation. Les développeurs utilisent généralement le code le plus à jour pour y repérer les bogues et, connaissant bien le projet, ils peuvent éviter les fonctionnalités encore instables. Il est courant pour eux de mettre très fréquemment à jour leur logiciel, et lorsqu'ils publient une modification du code, ils estiment raisonnable d'attendre des autres développeurs qu'ils l'aient intégrée à leur version du logiciel dans les 24 heures qui suivent.

Comment, alors, le projet doit-il s'y prendre pour proposer une version finalisée ? Doit-il, par exemple, se contenter d'utiliser une image de l'arborescence à un moment précis, d'en faire un beau paquet binaire et de l'offrir au monde en annonçant « voici la version 3.5.0 » ? Non évidemment ! D'abord, l'arborescence de développement ne sera jamais complètement propre et publiable. La branche peut contenir diverses nouvelles fonctionnalités en chantier. Quelqu'un pourrait avoir publié un correctif important pour un bogue précis, mais qui est controversé et encore débattu au moment où vous figez le programme. Si tel est le cas,

repousser la livraison, jusqu'à ce que le débat prenne fin, n'aiderait en rien. Un autre débat, pas forcément corrélé, pourrait naître entre-temps et vous devriez, dans ce cas, patienter jusqu'à sa fin, au risque d'attendre indéfiniment.

Dans tous les cas, figer l'arborescence entière entrerait forcement en conflit avec le développement en cours, même si l'arborescence courante était publiable. Si son image devient la version *3.5.0*, l'image suivante, vraisemblablement la *3.5.1*, contiendra principalement des correctifs des bogues introduits dans la version *3.5.0*. Mais si les deux sont des images de la même arborescence supposée faite par les développeurs entre deux versions ? Ils n'ont pas l'autorisation d'ajouter de nouvelles fonctionnalités, les directives de compatibilité les en empêchent. Or, tout le monde n'est pas motivé par l'écriture de correctifs pour les bogues de la version *3.5.0*. Certains pourraient désirer achever de nouvelles fonctionnalités et seraient irrités que la seule possibilité qu'on leur laisse est de choisir entre se tourner les pouces ou travailler sur ce qui ne les intéresse pas, simplement parce que le processus de publication du projet impose à l'arborescence de développement de rester dans un état de stabilité artificielle.

Les branches de publication sont la solution à ces problèmes. Ces branches sont utilisées au sein d'un logiciel de gestion de versions (voir branche) pour que le code destiné à être publié puisse être isolé du développement principal. Les logiciels libres ne sont pas les seuls à emprunter cette méthode, beaucoup d'équipes de développement de logiciels propriétaires font de même. Mais dans les environnement propriétaires, ces dernières sont parfois considérées comme un luxe, une « bonne pratique », dont on peut, compte tenu de contraintes de temps, se dispenser lorsque l'ensemble des équipes travaillent à la stabilisation de l'arborescence principale.

En revanche, les branches de publication sont incontournables dans le cadre d'un projet libre. J'ai vu des projets publier sans y avoir recours, mais le résultat est toujours le même : une partie des développeurs n'a rien à faire, alors que les autres, en général une minorité, s'échinent à sortir enfin la version. Le résultat est en général mauvais pour plusieurs raisons. D'abord, l'effort de développement général est ralenti. Ensuite, la qualité de la version s'en ressent puisque seules quelques personnes y travaillant sont pressées de terminer pour que tout le monde puisse se remettre au travail. Enfin, cette situation, où les différents types de travaux interfèrent, crée une fracture psychologique entre les équipes. Les développeurs qui se tournent les pouces aimeraient probablement contribuer aux branches de publication, lorsqu'ils en ont le temps et l'envie. Mais sans la branche, leur choix devient « Vais-je participer au code aujourd'hui ou non ? » plutôt que « Vais-je travailler aujourd'hui sur la version en instance ou sur la nouvelle fonctionnalité que j'ai amorcée dans le code principal ? »

Fonctionnement des branches de publication

Le processus exact de création d'une branche de publication dépend, bien entendu, de votre logiciel de gestion de versions, mais les concepts généraux sont les mêmes pour la plupart des systèmes. Une branche est en général issue d'une autre branche ou du tronc. Le tronc désigne généralement la partie où se fait le gros du développement, loin des contraintes liées aux sorties. La première branche de publication, celle qui mène à la sortie de la version « 1.0 », provient du tronc. Dans CVS, la commande de branche ressemble à ceci :

```
$ cd trunk-working-copy
$ cvs tag -b RELEASE_1_0_X
```

ou avec Subversion à cela :

```
$ svn copy http ://.../repos/trunk http ://.../repos/branches1.0.x
```

(Ces exemples impliquent un modèle de numérotation de version à 3 chiffres. L'espace faisant défaut ici pour fournir les commandes exactes de chaque logiciel de gestion de versions, j'utiliserai des exemples tirés de CVS et de Subversion en espérant que les commandes correspondantes dans les autres systèmes pourront aisément en être déduites.)

Notez que nous avons créé la branche *1.0.x* (la lettre x) plutôt que *1.0.0*. C'est parce que la même ligne mineure, c'est à dire la même branche, va être employée pour toutes les micro publications dans cette ligne. Le véritable processus de stabilisation d'une branche, en vue de sa sortie, est discuté dans la section appelée « Stabiliser une version » plus loin dans ce chapitre. Nous nous concentrons ici uniquement sur l'interaction entre le logiciel de gestion de versions et le processus de publication. Lorsque la branche est stabilisée et prête, c'est le moment d'en faire une image instantanée :

```
$ cd RELEASE_1_0-working-copy
$ cvs tag RELEASE_1_0_0
```

ou

```
$ svn copy http ://.../repos/branches/1.0.x http ://.../repos/tags/1.0.0
```

Cette étiquette représente l'état exact de l'arborescence source du projet pour la publication de la version *1.0.0* (c'est utile au cas où quelqu'un aurait besoin de restaurer une ancienne version, après que les paquets et binaires ne sont plus disponibles). La micro publication suivante, dans la même ligne, est préparée également sur la branche *1.0.x* et lorsqu'elle est prête, une étiquette est créée pour la version *1.0.1*. Répétez l'opération pour la version *1.0.2* et ainsi de suite… Lorsque le moment est venu de publier une version *1.1.x*, créez une nouvelle branche partant du tronc :

```
$ cd trunk-working-copy $ cvs tag -b RELEASE_1_1_X
```

ou

```
$ svn copy http ://.../repos/trunk http ://.../repos/branches/1.1.x
```

La maintenance des versions *1.0.x* et *1.1.x* se fait en parallèle, et les sorties peuvent être indépendantes pour chaque ligne. Il est courant de publier de nouvelles versions presque simultanément pour deux lignes différentes. L'ancienne série est recommandée aux administrateurs plus conservateurs qui ne veulent pas sauter le pas vers la version *1.1* sans s'y être consciencieusement préparés. Mais les personnes plus aventureuses peuvent également disposer d'une version plus récente (possédant la numérotation la plus élevée), afin de profiter des dernières fonctionnalités, même au prix d'une plus grande instabilité.

Ce n'est naturellement pas la seule stratégie de publication de branches. Et elle n'est pas forcément la meilleure non plus selon les circonstances, bien qu'elle ait fait ses preuves dans bien des projets auxquels j'ai participé. Choisissez la stratégie qui vous convient le mieux, mais souvenez-vous des points essentiels : une branche de publication sert à isoler le travail à publier de l'agitation du développement quotidien et à structurer le processus de publication. Ce dernier est décrit plus en détails dans la prochaine partie.

3. Stabiliser une version

La stabilisation consiste à rendre publiable une branche, c'est à dire décider des changements qui seront inclus dans cette version et ainsi élaborer son contenu.

Le terme « décider » est polémique. La course à la fonctionnalité de dernière minute est un phénomène courant pour les projets de logiciels collaboratifs : aussitôt que les développeurs s'aperçoivent qu'une sortie est imminente, ils se bousculent pour terminer leur travaux en cours et ne pas rater le train. C'est évidemment exactement ce que vous voulez éviter au moment de sortir une nouvelle version. Il serait nettement préférable qu'ils continuent à travailler sur ces fonctionnalités à leur rythme, sans se préoccuper du fait que leurs modifications seront incluses dans la version en préparation ou dans la suivante. Plus vous essaierez de surcharger une version à la dernière minute, plus le code sera instable et plus vous introduirez de bogues.

La plupart des ingénieurs logiciels s'accordent sur des critères de base permettant de décider quels types de changements devraient être autorisés lors de la période de stabilisation d'une version. Bien sûr, les correctifs de bogues graves sont autorisés, en particulier ceux qui ne possèdent pas de contournements. Les mises à jour de la documentation sont acceptables, tout comme les corrections de messages d'erreur (sauf lorsqu'ils sont considérés comme faisant partie de l'interface et doivent rester stables). De nombreux projets autorisent également, durant la stabilisation, certains changements peu risqués ou sans impact sur le cœur du programme et peuvent avoir des critères stricts pour mesurer les risques. Mais quel que soit le niveau de formalisme, le jugement humain reste indispensable. Il reste toujours des cas où le projet doit prendre la décision d'accepter ou non un changement. Comme chacun souhaite voir ses modifications préférées être incorporées à la nouvelle version, nombreux sont ceux qui voudront autoriser les modifications, tandis que ceux voulant les bloquer le sont nettement moins, voilà le risque.

Par conséquent, le moteur du processus de stabilisation d'une version est le « non ». Le plus délicat, en particulier pour les projets libres, est de trouver comment refuser tout en évitant de blesser ou de décevoir les développeurs et sans pour autant poser d'entraves aux modifications méritant d'être adoptées. Différents moyens le permettent, et il est assez simple de créer des systèmes satisfaisant à ces critères une fois que l'équipe les a identifiés. Je ferai ici une brève description de deux systèmes parmi les plus populaires malgré leurs différences, mais n'hésitez pas à vous montrer plus créatif : de nombreuses autres possibilités existent, il s'agit de deux exemples simples dont j'ai pu constater l'efficacité en pratique.

La dictature du propriétaire de version

Le groupe désigne un propriétaire de version. Cette personne décide finalement des changements qui seront inclus ou non dans la version. Bien sûr, il est normal, et attendu, qu'il y ait des discussions et des débats, mais le groupe doit accorder une autorité suffisante au propriétaire de version pour qu'il prenne les décisions finales. Pour que ce système fonctionne, la personne choisie doit disposer des compétences techniques requises pour comprendre toutes les modifications, mais aussi avoir le charisme et les compétences nécessaires pour mener la discussion et parvenir à la publication sans froisser excessivement les susceptibilités.

Le propriétaire de version est souvent amené à dire par exemple : « Je ne pense pas qu'il y ait de problème avec cette modification, mais nous n'avons pas assez de temps pour la tester maintenant, on ne devrait donc pas l'inclure dans cette version ». S'il possède une connaissance technique générale du projet, il serait judicieux de sa part d'expliquer en quoi le changement pourrait impacter la version (par exemple, par ses interactions avec d'autres parties du logiciel, ou pour une question de portabilité). Certains demanderont parfois des justifications, ou affirmeront que leurs modifications ne sont pas si risquées. Ces conversations ne seront pas conflictuelles tant que le propriétaire de version sera en mesure de peser tous les arguments, objectivement et sans camper systématiquement sur ses positions.

Notez que le propriétaire de version n'est pas nécessairement le chef du projet (si tant est qu'il y en ait un, voir la section « Les dictateurs bienveillants » du chapitre 4). Il est souvent préférable que ce ne soit pas une seule et même personne qui endosse les deux rôles. Les aptitudes requises sont assurément différentes. Pour une chose aussi importante que le processus de publication, il est sage d'apporter un contrepoids au jugement du chef de projet.

Comparez le rôle du propriétaire de version à celui, moins dictatorial, décrit dans la section « Contrôleur de version » plus loin dans ce chapitre.

Le choix des évolutions par vote

À l'extrême opposé du concept de dictature du propriétaire de version, les développeurs peuvent simplement voter pour décider des évolutions à inclure dans la version. Cependant, la stabilisation n'étant pas une opération portes ouvertes, il est important de concevoir le système de vote en gardant à l'esprit que l'ajout de modifications doit exiger l'implication de plusieurs développeurs. Une simple majorité devrait être une condition nécessaire, mais non suffisante, à un ajout (voir la section « Qui vote ? » dans le chapitre 4). Dans le cas contraire, un vote « pour » et aucun vote « contre » une modification suffirait à la valider et une dynamique indésirable se mettrait en place : chaque développeur voterait pour son propre changement et serait peu enclin à voter contre les modifications des autres par peur de représailles. Pour éviter cela, le système devrait favoriser la coopération entre sous-groupes de développeurs militant pour un changement. Non seulement plus de gens vérifient alors les modifications, mais en plus, les développeurs, en tant qu'individus, sont davantage enclins à voter contre. Ils savent, en effet, que personne, en particulier parmi ceux qui ont voté « pour », ne prendra leur vote « contre » comme un affront personnel. Plus il y a de personnes impliquées, plus la discussion se focalise sur les changements et non plus sur les individus.

Le système que nous employons dans le projet Subversion semble avoir atteint un bon équilibre, je le recommande donc. Pour qu'une modification soit appliquée à une branche de publication, au moins trois développeurs doivent voter en sa faveur et aucun contre. Un seul vote « contre » est suffisant pour que le changement ne soit pas ajouté, c'est à dire qu'un vote « contre » équivaut à un veto (voir la section intitulée « Vetos »). Naturellement, un tel vote doit être justifié et, en théorie, le veto peut être outrepassé si suffisamment de personnes pensent qu'il est excessif et forcent un vote spécial pour le contourner. En pratique, ceci ne s'est jamais produit, et je ne pense pas que ce soit le cas dans l'avenir. Les participants deviennent naturellement plus conservateurs quand on s'approche d'une publication et lorsque quelqu'un est suffisamment déterminé pour opposer son veto à une modification, ses raisons sont généralement bonnes.

La procédure de publication étant délibérément biaisée en faveur du conservatisme, les justifications données pour les vetos sont parfois plus procédurières que techniques. Par exemple, une personne peut être convaincue qu'une modification est correcte et ne causera probablement aucun nouveau bogue et voter pourtant contre son inclusion dans une version de mise à jour intermédiaire, simplement parce que cette modification est trop lourde, ou qu'elle apporte peut-être de nouvelles fonctionnalités ou risque de ne pas remplir totalement les règles de compatibilité ascendante. J'ai même déjà vu, à une occasion, des développeurs opposer leur veto simplement parce qu'ils avaient le pressentiment que la modification nécessitait des tests plus poussés, sans toutefois détecter le moindre bogue lors des relectures. Malgré les récriminations, le veto fut respecté et la modification n'a pas été incluse dans cette version (d'ailleurs, je ne me souviens pas si des bogues ont finalement été découverts ou non).

Gérer une stabilisation collégiale

Si votre projet opte pour un système de vote, vous devez rendre aussi accessibles que possible les outils du vote ainsi que le vote en lui-même. Bien qu'il existe de nombreux logiciels Open Source créés pour faciliter le vote électronique, un simple fichier texte (que l'on nommera par exemple « STATUTS » ou « VOTES ») dans la branche de publication se révèlera plus pratique. Le fichier, éditable par tous les développeurs, liste chaque évolution proposée ainsi que les votes pour ou contre. Les votes sont fréquemment accompagnés de notes et commentaires. Toutefois, proposer une modification ne signifie pas nécessairement voter pour, bien que les deux aillent souvent de pair. Voici un exemple de ce à quoi peut ressembler ce type de fichier :

> * r2401 (problème #49) Éviter la redondance de la connexion client/serveur. Justification : Supprime des accès réseaux inutiles ; changement minime et facile à vérifier. Notes : Sujet discuté sur http://.../mailing-lists/message-7777.html et autres messages dans ce sujet. Votes : +1 : jsmith, kimf -1 : tmartin (viole la compatibilité avec certains serveurs pre-1.0 ; il est vrai que ces serveurs sont bogués, mais il faut préserver la compatibilité autant que faire se peut)

Ici, la modification a reçu deux votes pour mais *tmartin* a opposé son veto, et l'a justifié dans une note entre parenthèses. La forme exacte de cette entrée importe peu : l'explication de *tmartin* pourrait se trouver dans la section « Notes » ou dans la description de la modification marquée par un titre « Description » pour correspondre aux autres sections… Il faut principalement vous assurer que toutes les informations nécessaires à l'évaluation soient disponibles, et que le système de vote soit aussi simple que possible. La modification soumise à débat est désignée par son numéro de révision dans le dépôt de sources (dans ce cas, il contient une seule révision, la r2401, mais une modification peut en référencer plusieurs). La révision pointe implicitement vers un changement du tronc puisqu'une modification issue de la branche de publication n'aurait pas eu à être soumise au vote. Si votre logiciel de gestion de versions ne dispose pas d'une syntaxe explicite pour se référer à des modifications individuelles, alors, le projet devrait en prévoir une. Pour que le vote soit fonctionnel, chaque modification doit être identifiable sans ambiguïté.

Ceux qui ont proposé ou voté pour une modification doivent s'assurer que cette dernière s'applique correctement et sans conflit à la branche de publication (voir Conflits). Si des conflits subsistent, la modification doit pointer vers un correctif ou vers une branche temporaire proposant une version ajustée de la modification, par exemple :

> *r13222, r13223, r13232 Réécrire l'algorithme de fusion automatique libsvn_fs_fs Justification : Performances inacceptables (>50 minutes pour un petit ajout) dans un dépôt contenant 300 000 révisions Branche : 1.1.x-r13222@13517 Votes : +1 : epg, ghudson

Cet exemple est authentique, il provient du fichier *STATUTS* utilisé lors de la publication de Subversion 1.1.4. Notez qu'il utilise les révisions originales comme identifiants uniques des modifications, malgré l'utilisation d'une branche contenant une version adaptée de la modification (la branche combine les trois révisions du tronc en une seule, la r13517, afin de faciliter sa fusion si elle est acceptée). Les révisions originales sont présentes pour faciliter leur consultation car les commentaires originaux y sont liés. La branche temporaire ne doit pas reprendre ces derniers pour éviter de dupliquer l'information (voir la section « Unicité de l'information » dans le Chapitre 3, Infrastructure Technique), le commentaire de la révision r13517 devrait simplement contenir « Ajuster r13222, r13223 et r12232 pour portage à la branche 1.1.x. ». Toute autre information peut être retrouvée dans les révisions originales.

Le responsable de version

La fusion (voir fusion ou portage) des modifications approuvées vers la branche de publication peut être effectuée par les différents développeurs. Un poste spécifique, dédié aux fusions, n'est pas indispensable. Si la charge de travail devient trop conséquente, il vaut mieux la répartir entre plusieurs développeurs.

Malgré le caractère décentralisé des votes et des fusions, une ou deux personnes guident le processus de publication. On utilise parfois le terme formel de « responsable de version » pour ce rôle sensiblement différent du « propriétaire de version » (voir la section « La dictature du propriétaire de version » précédemment dans ce chapitre) lequel garde le dernier mot sur les changements à appliquer. Les responsables de version effectuent le suivi des

modifications actuellement en cours de traitement : combien ont été approuvées, combien sont en voie d'obtenir l'approbation, etc. C'est leur rôle aussi de suggérer subtilement aux développeurs d'examiner certains changements ne bénéficiant pas de l'attention qu'ils méritent. Lorsqu'un lot de modifications est approuvé, ils se chargent (en général de leur propre initiative) de les insérer dans la branche de publication. Les autres développeurs doivent cependant comprendre que tout ce travail n'incombe pas aux seuls responsables de version, sauf prérogatives explicites. Lors de la publication de la version (voir la section « Tests et sortie » plus loin dans ce chapitre), les responsables de version sont également responsables de la création des paquets définitifs, de la collecte des signatures numériques, de la mise à disposition des paquets et de l'annonce publique.

4. La création de paquets

Les logiciels libres sont traditionnellement distribués sous la forme de code source, interprété (Perl, Python, PHP, etc.) ou nécessitant une compilation préalable (C, C++, Java, etc.). Les logiciels compilés ne seront généralement pas construits par les utilisateurs eux-mêmes, mais installés sous la forme d'un paquet binaire prêt à l'exécution (voir la section appelée « Paquets binaires » plus loin dans ce chapitre). Ces paquets sont néanmoins toujours issus d'une source de distribution de référence. Le paquet source doit porter un identifiant de version le rendant parfaitement identifiable. Un paquet nommé « Scanley 2.5.0 » désigne en fait « l'arborescence de fichiers sources constituant Scanley 2.5.0 une fois compilée (si nécessaire) et installée ».

Cette arborescence répond à des normes plutôt strictes qui sont, à quelques rares exceptions près, toujours suivies. Sauf cas de force majeure, votre projet devrait lui aussi suivre ces normes.

Le format

Le code source devrait être mis à disposition dans le format standard du transport d'arborescences de répertoires. Pour les systèmes d'exploitation Unix ou dérivés, la convention est le format TAR, compressé par compress, gzip, bzip ou bzip2. Pour MS Windows, c'est le format zip. Ce dernier assure également la compression, ce qui élimine le besoin de compresser l'archive après sa création.

Les fichiers TAR
TAR signifie « Tape ARchive », car le format tar représente une arborescence de répertoires comme un flux linéaire de données, ce qui le rend idéal pour sauvegarder des répertoires sur cassette. Cette même propriété en fait le standard idéal pour distribuer les arborescences sous la forme d'un fichier unique. La création de fichiers tar compressés (ou tarballs) est assez facile. Sur certains systèmes, la commande tar peut produire elle-même des archives compressées alors que sur d'autres, un programme de compression distinct est nécessaire.

Nom et structure

Le nom du paquet est composé du nom du logiciel suivi du numéro de version et du suffixe du format correspondant au type d'archive. Par exemple, le nom pour Scanley 2.5.0, pour Unix, compressé avec GNU Zip (gzip) ressemble à ceci :

scanley-2.5.0.tar.gz

ou pour Windows, en utilisant une compression de type zip :

scanley-2.5.0.zip

Ces deux archives, une fois décompressées, créent un nouveau dossier nommé *scanley-2.5.0* dans le dossier courant. Dans ce nouveau dossier, le code source est prêt à être compilé (si une compilation est requise) et à être installé. À la racine du nouveau dossier se trouve un fichier texte nommé *README* (ou *LISEZ-MOI*) expliquant le but du logiciel, sa version ainsi que d'autres ressources comme le site Web du projet, d'autres fichiers pertinents, etc. Parmi ces autres fichiers, on trouve un fichier *INSTALL*, semblable au fichier *README*, qui fournit les instructions permettant de construire et d'installer le logiciel pour tous les systèmes d'exploitation pris en charge. Comme nous l'avons déjà évoqué dans la section « Comment appliquer une licence à votre logiciel » du chapitre 2, on trouve également un fichier *COPYING* ou *LICENCE*, précisant les conditions de distribution du logiciel.

On doit également y trouver un fichier *CHANGES* ou *MODIFICATIONS* (parfois appelé *NEWS* ou *NOUVEAUTÉS*) reprenant les améliorations apportées par cette version. Le fichier *CHANGES* liste les modifications portant sur toutes les versions, dans l'ordre antichronologique, afin que les modifications de la version courante apparaissent en premier. Compléter cette liste est, en général, la dernière chose devant être effectuée sur une branche de publication en cours de stabilisation. Certains projets rédigent cette liste au fur et à mesure du développement, d'autres préfèrent conserver cette étape pour la fin et confier sa rédaction à une personne. Les modifications sont tirées des journaux de gestion de versions et ressemblent à cet extrait :

> Version 2.5.0 (20 décembre 2004) depuis /branches/2.5.x) http ://svn.scanley.org/repos/svn/tags/2.5.0/ Nouvelles fonctionnalités, améliorations : * Ajout des requêtes d'expressions régulières * Ajout du support UTF-8 et UTF-16 pour les documents * Documentation traduite en polonais, russe * et malgache * ... Correctifs : * Correction du bogue de ré-indexation (bogue #945) * Correction de quelques bogues de requêtes (bogues #815, #1007, #1008)

La liste peut être aussi longue que nécessaire, mais il n'est pas indispensable d'y inclure chaque petite correction ou amélioration. Elle sert à fournir aux utilisateurs une vision générale de ce qu'ils gagneront à utiliser la nouvelle version. En fait, la liste des modifications apparaîtra en général dans l'e-mail d'annonce (voir la section appelée « Tests et sortie » plus loin dans ce chapitre), rédigez-la donc en gardant en tête sa future audience.

CHANGES et ChangeLog
Traditionnellement, un fichier nommé ChangeLog liste l'intégralité des changements apportés à un projet, c'est -à-dire chaque révision du logiciel de gestion de versions. Il existe différents formats pour les fichiers ChangeLog, mais les détails du format ne sont pas importants ici puisqu'ils contiennent tous les mêmes informations : la date de la modification, son auteur ainsi qu'un bref résumé (ou simplement le message de journal pour ce changement).
Un fichier CHANGES est différent. Il ne reprend que les modifications jugées importantes pour un certain public : certaines méta-données comme la date exacte ou l'auteur ont été supprimées. Pour éviter les confusions, n'intervertissez pas ces deux termes. Certains projets utilisent « NEWS » plutôt que « CHANGES ». Même si la confusion potentielle avec « ChangeLog » est ainsi évitée, le terme est mal choisi puisque le fichier CHANGES répertorie les modifications portant sur toutes les versions antérieures, et comprend donc beaucoup d'anciennes « nouveautés » en plus de celles figurant en haut du fichier.
Les fichiers ChangeLog semblent, de toute façon, en voie de disparition. Ils étaient utiles à l'époque où CVS était le logiciel de gestion de versions incontournable et dont il n'était pas simple d'extraire ces données. Les logiciels de gestion de versions plus modernes peuvent facilement restituer par requêtes les données qui étaient jadis inscrites dans le ChangeLog, ce qui rend inutile la maintenance d'un fichier statique contenant ces données, puisque le ChangeLog se résume alors à une réplique du journal de messages déjà enregistré dans le dépôt.

La structure du code source, dans l'arborescence, doit être aussi proche que possible de celle du projet telle qu'elle apparaît dans le dépôt de gestion de versions. Quelques différences peuvent exister, par exemple, le paquet peut contenir certains fichiers générés nécessaires à la configuration et à la compilation (voir la section nommée « Compilation et installation » plus loin dans ce chapitre), ou il peut contenir un programme tiers requis, mais qui n'est pas maintenu par le projet, et que les utilisateurs ne possèdent probablement pas encore. Mais, même si l'arborescence distribuée correspond exactement à une arborescence de développement dans le dépôt de gestion de versions, la distribution elle-même ne doit pas être une copie de travail (voir « Copie de travail » dans la section « Gestion de versions » du chapitre 3) et doit représenter un point de référence immuable, une configuration particulière et non-modifiable de fichiers sources. Une copie de travail pourrait être modifiée par l'utilisateur, lequel penserait alors disposer de la version officielle alors qu'il ne s'agirait que d'une version modifiée.

Souvenez-vous que le contenu est toujours le même, indépendamment du paquet. La version, c'est à dire l'entité précise à laquelle on fait référence quand on parle de « Scanley 2.5.0 », est l'arborescence créée lorsqu'on extrait les fichiers d'une archive zip ou tarball. Le projet devrait donc proposer tous ces fichiers au téléchargement :

scanley-2.5.0.tar.bz2 scanley-2.5.0.tar.gz scanley-2.5.0.zip

... mais l'arborescence issue de leur extraction doit être rigoureusement la même. Cette arborescence source est la distribution ; la forme sous laquelle elle est téléchargée est une simple question de commodité. Certaines différences triviales entre les paquets sources sont

possibles : par exemple, dans les fichiers textes de paquets Windows, les lignes devraient se terminer par CRLF (Carriage Return and Line Feed [1] tandis que les paquets Unix ne devraient utiliser que LF. Les arborescences peuvent être légèrement arrangées en fonction du système d'exploitation cible si ce dernier demande une structure spécifique pour la compilation. Il s'agit néanmoins de transformations triviales. Les fichiers sources de base devraient être identiques pour tous les paquets d'une version donnée.

De l'utilisation des majuscules

Les gens utilisent naturellement des majuscules pour les noms de projets, alors traités comme des noms propres. Il en est de même pour les acronymes : « MySQL 5.0 », « Scanley 2.5.0 », etc. C'est au projet de décider si le nom du paquet doit utiliser la même casse. *Scanley-2.5.0.tar.gz* ou *scanley-2.5.0.tar.gz* sont tous les deux valables par exemple (je préfère la dernière proposition pour ne pas forcer les gens à appuyer sur MAJ, mais de nombreux projets utilisent une majuscule dans le nom du paquet). Il faut néanmoins que le répertoire créé en extrayant le tarball utilise la même convention. Évitez les surprises, l'utilisateur devrait pouvoir prédire sans ambiguïté le nom exact du répertoire créé quand il extraira la distribution.

Pré-version

Quand vous rendez publique une pré-version ou une version candidate, le qualificatif fait concrètement partie du numéro de version, par conséquent ajoutez-le au nom du paquet. Par exemple, les phases successives alpha et bêta déjà abordées dans la section « Les composants d'un numéro de version » correspondraient aux noms de paquets suivants :

scanley-2.3.0-alpha1.tar.gz scanley-2.3.0-alpha2.tar.gz scanley-2.3.0-beta1.tar.gz scanley-2.3.0-beta2.tar.gz scanley-2.3.0-beta3.tar.gz scanley-2.3.0.tar.gz

Le premier paquet est extrait dans un répertoire nommé *scanley-2.3.0-alpha1*, le second dans un répertoire *scanley-2.3.0-alpha2* et ainsi de suite.

Compilation et installation

Lorsqu'une compilation ou une installation depuis la source sont nécessaires, il existe en général des procédures standardisées que les utilisateurs expérimentés connaissent. Par exemple, pour les programmes écrits en C, C++ ou certains autres langages compilés, la méthode standard sous Unix et dérivés est la suivante :

```
$ ./configure $ make # make install
```

1. NdT : retour du chariot et passage à la ligne.

La première de ces commandes détecte automatiquement autant d'informations que possible sur l'environnement et prépare l'étape de construction, la seconde commande construit le logiciel (mais ne l'installe pas) et la dernière commande l'installe sur le système. Les deux premières sont effectuées en tant qu'utilisateur simple, la troisième en tant qu'administrateur. Pour plus de détails à propos de la mise en place de ce système, je vous renvoie à l'excellent livre *GNU Autoconf, Automake et Libtool* par Vaughan, Elliston, Tromey et Taylor. Il est publié en tant que « treeware » par New Riders et son contenu est accessible librement en ligne [1].

Ce n'est pas le seul standard qui existe, mais c'est le plus répandu. Le système de construction ANT [2] gagne en popularité, particulièrement auprès des projets écrits en Java, et possède ses propres standards de procédures pour la construction et l'installation. Certains langages de programmation, tels que Perl et Python, recommandent également une méthode standard pour les programmes écrits dans ces langages (par exemple, les modules Perl emploient la commande *perl Makefile.pl*). Si les standards applicables à votre projet ne vous paraissent pas évidents, demandez à un programmeur expérimenté ; il existe très probablement un standard que vous ne connaissez pas encore.

Choisissez le standard le plus adapté à votre projet et ne changez qu'en cas de nécessité absolue. Les procédures d'installation standard sont devenues des réflexes conditionnés pour beaucoup d'administrateurs système. Des commandes familières, dans le fichier *INSTALL*, les rassureront sur l'importance que vous accordez aux standards, et ils auront une meilleure approche *a priori* de votre logiciel. De même, comme nous l'avons vu dans la section « Téléchargements » du chapitre 2, utiliser une séquence de construction standard plaît aux développeurs potentiels.

Sous Windows, les standards de construction et d'installation ne sont pas aussi bien définis. Pour les projets nécessitant une compilation, il semblerait que l'habitude soit de créer une arborescence s'intégrant à l'espace de travail des environnements de développement standards de Microsoft (Developper Studio, Visual Studio, VS.NET, MSVC++, etc.). Selon la nature de votre logiciel, il vous sera possible de proposer une option de construction semblable à celle d'Unix sur Windows grâce à l'environnement Cygwin [3]. Et bien sûr, si vous utilisez un langage ou framework qui possède ses propres conventions comme Perl ou Python, vous devriez simplement employer la méthode standard adéquate indépendamment du système d'exploitation Windows, Unix, Mac OS ou autres.

Soyez prêt à faire de nombreux efforts supplémentaires pour rendre votre projet conforme aux standards de construction et d'installation. Ce sont des étapes clés : les choses peuvent devenir plus complexes par la suite, si vraiment c'est nécessaire, mais il serait fortement dommageable pour l'image de votre projet que les utilisateurs aient à fournir plus d'efforts que prévus dès ce premier contact avec le logiciel.

1. http ://sources.redhat.com/autobook/
2. http ://ant.apache.org/
3. http ://www.cygwin.com/

Paquets binaires

Les paquets embarquant le code source constituent le format canonique d'une version, mais la plupart des utilisateurs installeront un paquet binaire, fourni par le système de distribution de logiciels de leur système d'exploitation ou téléchargé manuellement depuis le site Web du projet ou d'un tiers. « Binaire » ne signifie pas forcément « compilé », cela veut simplement dire que le paquet est pré-configuré de sorte que l'utilisateur puisse l'installer sur son ordinateur sans avoir à passer par les procédures habituelles de construction/installation du fichier source. On trouve sous Redhat GNU/Linux le système RPM, sous Debian GNU/Linux le système APT (*.deb*), sur MS Windows en général des fichiers *.MSI* ou des fichiers d'installation automatique *.exe*.

Qu'ils viennent d'un proche du projet ou d'une personne qui y est complètement étrangère, ces paquets sont officiels et les problèmes qu'ils rencontrent seront ajoutés au système de suivi de bogues. Le projet a donc tout intérêt à travailler en étroite collaboration avec ceux qui préparent les paquets et à leur fournir des directives claires afin que les paquets soient fidèles au logiciel.

Les assembleurs doivent impérativement baser leurs paquets binaires sur une version officielle du code source. Ils seront parfois tentés d'extraire une version plus évoluée du dépôt ou d'y inclure des modifications plus récentes pour apporter aux utilisateurs certaines corrections de bogues ou d'autres améliorations. En agissant ainsi, ils pensent rendre service en proposant le code le plus récent, mais gare à la confusion. Les projets savent traiter les rapports concernant les bogues rencontrés dans les versions publiées, dans les ajouts récents au tronc ou dans les branches principales du code (des bogues trouvés par ceux qui font tourner délibérément le code le plus récent). Quand un rapport de bogue est enregistré pour une version ou une branche précise, la personne traitant le rapport sera dans la plupart des cas capable de confirmer la présence du bogue et, s'il a été corrigé, il pourra alors conseiller à l'utilisateur de mettre son logiciel à jour ou d'attendre la prochaine version. Et enfin, si le bogue n'est pas encore répertorié, son classement nécessite de connaître la version affectée.

Mais les projets ne sont pas préparés à recevoir des rapports de bogues basés sur des versions mal définies ou hybrides. Ces bogues sont très difficiles à reproduire et peuvent être le résultat d'interactions imprévisibles entre des changements isolés extraits d'un développement plus récent. Ils peuvent donc causer des comportements inattendus sans que les développeurs du projet en soient responsables. Ces rapports particuliers peuvent faire perdre un temps considérable à votre projet, j'en ai moi-même été témoin : quelqu'un rencontrait un bogue avec une version corrigée du code source officiel. Personne dans l'équipe de développement ne parvenait à reproduire le bogue et chacun a dû chercher en vain l'origine de ce comportement inexplicable.

Certains assembleurs insisteront toujours pour inclure certaines modifications du code en développement. Il faut les encourager à avertir les développeurs du projet de leurs intentions. Même si les développeurs ne le voient pas d'un bon œil, ils seront au moins prévenus en cas de rapport de bogue inattendu. Ils pourront ainsi publier une mise en garde sur le site du projet et demander au créateur du paquet d'en faire de même à un endroit bien en vue afin que les personnes utilisant ce paquet binaire sachent que ce qu'ils ont entre les mains n'est pas exactement identique à ce que le projet a officiellement publié. Même si c'est difficile, il faut éviter d'envenimer la situation. Les assembleurs ne partagent simplement

pas la vision des développeurs. Ils cherchent principalement à apporter aux utilisateurs le meilleur du logiciel. C'est aussi le but des développeurs bien entendu, mais ces derniers doivent également maîtriser la version exacte des paquets utilisés pour rendre les rapports de bogues cohérents et pour assurer la compatibilité de leur travail. Ces objectifs entrent parfois en conflit. Il faut alors se souvenir que le projet n'a aucun contrôle sur le travail des assembleurs et que chacun a des obligations envers l'autre. En effet, le projet produit généreusement le code mais les assembleurs, de leur côté, effectuent un travail souvent ingrat pour rendre le projet largement accessible et l'ouvrir à une communauté que le projet seul n'aurait peut être pas atteinte. Vous pouvez vous trouver en désaccord, mais restez courtois pour que tout se passe pour le mieux.

5. Tests et sorties

Une fois le tarball des sources créé à partir d'une branche de publication stabilisée, la partie visible de la publication commence. Mais avant que le tarball soit mis à la disposition du monde entier, il doit être testé et approuvé par un minimum de développeurs, en général trois ou plus. Cette validation ne sert pas seulement à chercher des bogues évidents : idéalement les développeurs téléchargent l'archive, la construisent et l'installent sur un système propre, puis effectuent l'ensemble des tests de non-régression (voir la section appelée « Tests automatisés » dans le chapitre 8) et quelques tests manuels. En supposant que l'archive passe ces épreuves, ainsi que toute autre série de tests que le projet pourrait avoir fixée, les développeurs signent numériquement l'archive en utilisant GnuPG [1], PGP [2], ou tout autre programme capable de produire une signature compatible PGP.

Dans la plupart des projets, les développeurs utilisent leurs propres signatures numériques plutôt qu'une clé partagée par le projet. Tous les développeurs qui le souhaitent peuvent apposer leur signature (il y en a donc un nombre minimum, mais pas de maximum). Un grand nombre de signatures prouve que le programme a fait l'objet de nombreux tests, ce qui augmente la confiance des utilisateurs avertis.

Une fois approuvés, tous les fichiers (c'est à dire, les tarballs, les fichiers zip et autres formats de paquet) devraient être placés dans la section téléchargement du projet, accompagnés d'une signature numérique et d'une empreinte MD5/SHA1 (voir l'article Wikipédia sur la fonction de hachage [3]). Plusieurs standards existent. Une solution consiste à mettre à disposition, avec le paquet, le fichier de signature numérique correspondante et le fichier d'empreinte. Par exemple, pour le paquet *scanley-2.5.0.tar.gz*, placez dans le même répertoire, un fichier *scanley-2.5.0.tar.gz.asc* contenant la signature numérique de l'archive, un fichier *scanley-2.5.0.tar.gz.md5* contenant l'empreinte MD5 et, à votre convenance, un fichier *scanley-2.5.0.tar.gz.sha1* contenant l'empreinte SHA1. Il est également possible de rassembler toutes les signatures pour tous les paquets proposés dans un fichier unique, *scanley-2.5.0.sigs*, et de faire de même avec les empreintes.

Peu importe la solution que vous retiendrez tant que le système est simple, documenté et cohérent entre les publications. Le but de toutes ces signatures et empreintes est de permettre

1. http ://www.gnupg.org/
2. http ://www.pgpi.org/
3. http ://fr.wikipedia.org/wiki/Fonction_de_hachage

aux utilisateurs de vérifier que la copie reçue n'a pas été falsifiée à des fins malveillantes. Les utilisateurs s'apprêtent à faire tourner ce code sur leur ordinateur, or, s'il a été altéré, une personne mal intentionnée peut ainsi créer une porte dérobée pour accéder à toutes leurs données. Dans la section « Les mises à jour de sécurité », plus loin dans ce chapitre, nous reviendrons sur la paranoïa.

Versions candidates

Pour les versions introduisant beaucoup de changements, de nombreux projets publient une version candidate (Release Candidate ou RC), par exemple *scanley-2.5.0-beta1* avant *scanley-2.5.0*. Le but d'une version candidate est de soumettre le code à un maximum de testeurs avant de le labelliser « version officielle ». Si des problèmes apparaissent, ils sont corrigés dans la branche de publication, et une nouvelle version candidate est proposée (*scanley-2.5.0-beta2*). Le cycle se poursuit jusqu'à ce qu'il n'y ait plus de bogues rédhibitoires. La version candidate devient alors la version officielle. Tout ce qui différencie la dernière version candidate de la version officielle est donc le qualificatif qui est retiré du numéro de version.

Pour le reste, une version candidate doit être traitée comme une version finale. Les qualificatifs *alpha*, *beta* ou *rc* suffisent à avertir les utilisateurs conservateurs de patienter jusqu'à la version finale et les e-mails d'annonce de la version candidate doivent mettre en avant qu'elle a pour but d'obtenir des retours de la part des utilisateurs. Sinon, accordez aux versions candidates la même attention qu'aux versions officielles car leur mise à disposition est la meilleure façon de trouver de nouveaux bogues, et parce que vous ne savez jamais, après tout, quelle version candidate deviendra la version officielle.

Annoncer une publication

Annoncer une publication, c'est comme annoncer n'importe quel autre évènement et les procédures décrites dans la section « Les annonces » du chapitre 6 s'appliquent. Il y a cependant quelques points particuliers à prendre en compte dans cette situation.

À chaque fois que vous fournissez l'URL d'un tarball, assurez-vous de fournir également les empreintes MD5/SHA1 et des liens vers les fichiers contenant les signatures numériques. Puisque l'annonce se fera sur plusieurs médias (listes de diffusion, pages d'informations, etc.), les utilisateurs peuvent obtenir les empreintes depuis différentes sources, ce qui donne aux utilisateurs les plus exigeants sur la sécurité une assurance supplémentaire que ces dernières n'ont pas été altérées. Proposer différents liens vers les fichiers de signatures numériques ne rend pas ces signatures plus sûres, mais démontre aux gens (en particulier ceux qui ne suivent pas le projet de près) que la sécurité est prise au sérieux.

Dans l'e-mail d'annonce et dans les pages d'information contenant plus qu'un simple argumentaire « commercial », assurez-vous d'inclure les passages pertinents du fichier *CHANGES* pour informer les utilisateurs des avancées apportées par cette mise à jour. Cela vaut autant pour les versions candidates que pour les versions finales, la présence de correctifs de bogues et de nouvelles fonctionnalités est importante pour pousser les gens à tester les versions candidates.

Pour finir, n'oubliez pas de remercier l'équipe de développement, les testeurs et toutes les personnes qui ont pris le temps de faire de bons rapports de bogues. Ne citez personne précisément, sauf si quelqu'un de particulier est à l'origine d'un travail singulièrement important, un travail que tout le monde a pu apprécier au sein du projet. Ne vous laissez pas entraîner sur la pente glissante de la surenchère de remerciements (voir la section « Remerciements » dans le chapitre 8).

6. Maintenir plusieurs versions

Les projets les plus matures maintiennent parallèlement plusieurs versions. Une fois publiée, la version *1.0.0* peut continuer à vivre via des micro versions (correctifs) *1.0.1*, *1.0.2*, etc. jusqu'à ce que le projet décide explicitement de mettre un terme à son support. Le simple fait de publier la version *1.1.0* n'est pas une raison suffisante pour clore la série des versions *1.0.x*. Certains utilisateurs se fixent comme règle de ne jamais utiliser une première version mineure ou majeure, laissant aux autres le soin de dénicher les bogues de la version *1.1.0* et attendent la version *1.1.1*. N'y voyez pas forcément une attitude égoïste (souvenez-vous qu'ils se privent ainsi des correctifs et des nouvelles fonctionnalités) : ils ont simplement décidé de ne prendre aucun risque avec les mises à jour à cause de contraintes spécifiques. De la même manière, si le projet découvre un bogue important dans la version *1.0.3*, peu de temps avant de sortir la version *1.1.0*, il serait sévère de n'effectuer la correction que dans la version *1.1.0* et de demander à tous les utilisateurs des versions *1.0.x* de se mettre à jour. Pourquoi ne pas publier à la fois les versions *1.1.0* et *1.0.4* pour contenter tout le monde ?

Une fois la ligne de version *1.1.x* bien avancée, vous pouvez déclarer officiellement la fin de support des versions *1.0.x*. Que cette annonce bénéficie d'un communiqué particulier ou qu'elle soit mentionnée lors de la sortie de la version *1.1.x* importe peu, les utilisateurs doivent être informés que l'ancienne ligne n'est plus supportée pour être en mesure de prendre des décisions sur une éventuelle mise à jour.

Certains projets fixent une durée de support pour les anciennes lignes. Dans un contexte Open Source, « supporter » signifie accepter les rapports de bogues et publier des versions de maintenance lorsque des bogues importants sont trouvés. D'autres projets ne se fixent pas de durées précises, mais s'appuient sur le nombre de rapports de bogue reçus pour évaluer le nombre d'utilisateurs de l'ancienne ligne. Lorsque la proportion descend sous un certain seuil, ils déclarent la fin de cette ligne et en stoppent le support.

À chaque publication, assurez-vous que le système de suivi de bogues propose bien les versions et jalons nécessaires pour permettre aux personnes rapportant les problèmes de les associer à la version adéquate. N'oubliez pas non plus de proposer un jalon « Développement » ou « Récent » pour la version de développement puisque certaines personnes (et pas uniquement des développeurs actifs) gardent souvent une longueur d'avance sur la version officielle.

Mises à jour de sécurité

La plupart des détails concernant la gestion des failles de sécurité ont été abordés dans la section « Annoncer les failles de sécurité » du chapitre 6, mais il nous reste à préciser certains points concernant la publication des mises à jour de sécurité.

Une mise à jour de sécurité est une version dédiée à la résolution d'une vulnérabilité. Le code qui résout le problème ne doit pas être rendu public avant la mise à disposition d'un correctif, ce qui signifie non seulement que le correctif ne peut être enregistré dans le dépôt avant le jour de sa sortie, mais également, que cette version ne peut faire l'objet de tests publics avant sa parution officielle. Les développeurs peuvent, bien entendu, examiner le correctif en interne et tester la version en privé, mais il est impossible de faire des tests à grande échelle.

Pour cette raison, une mise à jour de sécurité, par rapport à la version précédente, n'ajoute que le correctif de sécurité : aucune autre modification ne doit y figurer. La raison est simple : plus vous ajoutez de modifications non testées, plus vous augmentez le risque que l'une d'entre elles contienne un nouveau bogue, peut-être même un nouveau bogue de sécurité ! Ce conservatisme va aussi dans le sens des administrateurs qui pourraient avoir besoin de déployer le correctif de sécurité sans surprise.

Les mises à jour de sécurité vous contraindront, parfois, à quelques tours de passe-passe. Par exemple, le projet pourrait déjà travailler sur la sortie de la *1.1.3* (déjà publiquement annoncée) corrigeant quelques bogues de la version *1.1.2*, lorsqu'un rapport de sécurité arrive. Évidemment, les développeurs ne peuvent pas parler du problème de sécurité avant d'avoir mis un correctif à disposition. Ils doivent donc continuer à parler en public de la version *1.1.3* avec son contenu prévu initialement. Mais la version *1.1.3* réelle n'embarquera finalement que le correctif de sécurité et tous les autres changements devront être reportés à la version *1.1.4* (qui contiendra bien sûr le correctif de sécurité, comme toutes les versions ultérieures).

Vous pouvez utiliser le système de numérotation pour signaler les mises à jour de sécurité. Par exemple, le numéro de version *1.1.2.1* pourrait signaler qu'il s'agit d'une mise à jour de sécurité de la version *1.1.2* et que toutes les versions supérieures (*1.1.3*, *1.2.0*, etc.) contiennent le même correctif de sécurité. Pour les utilisateurs avertis, ce système fournit de nombreuses informations. Cela risque néanmoins de sembler étrange à ceux qui ne suivent pas le projet. Ils auront l'habitude de voir un numéro à trois chiffres la majeure partie du temps puis un quatrième apparaît de temps en temps. La plupart des projets que j'ai suivis choisissent la constance et utilisent simplement le numéro suivant pour les mises à jour de sécurité, même si cela implique de repousser d'un numéro les versions prévues.

7. Publications et développement quotidien

Le maintien de plusieurs versions simultanées a des conséquences sur le développement au quotidien. Le projet doit s'astreindre à une certaine discipline (recommandée de toute façon) : chaque modification n'apporte qu'un changement à la fois et des changements isolés ne sont jamais mélangés au sein d'un même *commit*. Si un changement est trop important ou trop impactant, il est conseillé de le diviser en plusieurs *commits* plus petits, chacun étant une sous-partie bien définie et n'embarquant rien qui ne soit relatif au changement global.

Voici un exemple de modification mal conçue :

> ———————————————— r6228 / toto / 30-06-2004 22 :13 :07 -0500 (Mercredi, 30 juin 2004) / 8 lignes Corrige le problème #1729 : Faire que l'indexation avertisse l'utilisateur quand un fichier est modifié lors de son indexation.
>
> * ui/repl.py (ChangingFile) : Nouvelle classe d'exception. (DoIndex) : Prise en charge de la nouvelle exception.
>
> * indexer/index.py (FollowStream) : Afficher une nouvelle exception si un fichier change durant l'indexation. (BuildDir) : Sans lien, retire des commentaires devenus obsolètes, refonte d'une partie du code et corrige les vérificateurs d'erreurs lors de la création d'un répertoire.
>
> Autres nettoyages sans lien :
>
> * www/index.html : Correction de quelques coquilles, date de la prochaine mise à jour ————————————————

Le problème apparaît dès que quelqu'un veut transposer le correctif portant sur le vérificateur d'erreur *BuildDir* dans une branche de maintenance. Les autres modifications ne sont pas souhaitées : le correctif du problème #1729 n'a pas forcément été approuvé pour la branche de maintenance et les améliorations de *index.html* sont ici sans objet. Mais il ne peut pas extraire facilement la modification apportée à *BuildDir* grâce aux outils de fusion du logiciel de gestion de versions puisque la modification groupe logiquement diverses modifications sans rapport. En fait, pas besoin d'attendre la fusion pour voir le problème apparaître. Le simple fait de lister la modification pour la soumettre au vote serait problématique : au lieu de simplement fournir le numéro de révision, la personne qui propose le vote devrait préparer un correctif spécial, ou changer de branche simplement pour isoler la partie du changement qu'il désire faire valider. Tout ceci rend le travail des autres plus difficile, uniquement parce que celui qui a enregistré ces modifications n'a pas pris la peine de les séparer en groupes logiques isolés.

Concrètement, ce *commit* unique aurait dû être séparé en quatre *commits* distincts : un pour le problème #1729, un autre pour retirer les commentaires devenus obsolètes et reformater le code dans *BuildDir*, un autre pour réparer la vérification d'erreurs dans *BuildDir* et le dernier portant sur les améliorations de *index.html*, le troisième étant le changement proposé pour la branche de la version de maintenance.

Les changements doivent être atomiques, pas seulement pendant la période de stabilisation pré-publication, mais tout le temps. Psychologiquement, un changement unifié autour d'un thème précis est plus facile à vérifier et à annuler si nécessaire (dans certains logiciels de gestion de versions, l'annulation est en fait un type particulier de fusion). Si chacun fait preuve d'un peu de discipline de gros problèmes ultérieurs peuvent être évités.

Planifier les publications

L'un des domaines où les projets Open Source diffèrent historiquement des projets propriétaires, est celui de la planification des publications. Les projets propriétaires ont, en général, des dates butoirs bien définies. Parfois parce qu'on a promis aux clients une mise à jour à une date précise, parce que la nouvelle version doit être coordonnée avec d'autres actions liées aux marketing, ou parce que les partenaires financiers qui ont investi dans le projet demandent à voir des résultats avant de s'engager plus avant. Les projets de logiciels libres, d'un autre côté, étaient encore récemment motivés principalement par l'« amateurisme » pris dans son sens le plus littéral : les développeurs écrivaient simplement par plaisir. Personne ne ressentait l'obligation de livrer le produit avant que toutes les fonctionnalités ne soient prêtes, et d'ailleurs pourquoi auraient-ils dû le faire ? Ce n'était pas comme si l'emploi de quelqu'un était en jeu.

De nos jours, de nombreux projets libres sont financés par de grands groupes et sont donc, de plus en plus, influencés par la culture d'entreprise et ses dates butoirs. C'est positif à bien des égards, mais cela peut créer des conflits entre les priorités des développeurs rémunérés et celles des bénévoles. La question de la date de publication et de son contenu cristallise souvent ces divergences. Les développeurs salariés, sous pression, voudront naturellement choisir une date pour la prochaine version et souhaiteront que l'activité de chacun s'aligne dessus. Mais les priorités des volontaires peuvent être bien différentes, peut-être souhaitent-ils achever des fonctionnalité ou quelques tests, ils estimeront donc que la sortie peut attendre.

Seuls la discussion et le compromis permettent de résoudre ce genre de problème. Mais vous pouvez limiter la fréquence et l'importance des frictions en dissociant version future et date de sortie. Essayez d'orienter la discussion sur les versions à court et moyen termes et sur les fonctionnalités que ces dernières devraient incorporer. N'avancez pas encore de date, sauf pour donner un ordre d'idée, et encore, avec une grande marge d'erreur [1]. Une fois les évolutions déterminées, les discussions autour de chaque version sont plus cadrées et donc moins sujettes à controverses. Vous créez en même temps une certaine inertie à vaincre pour quiconque souhaiterait proposer de nouvelles fonctionnalités ou complications. Si le périmètre d'une version est bien défini, il appartient à celui qui propose l'élargissement de le justifier, même si la date de publication n'a pas encore été fixée.

Dans sa biographie en plusieurs volumes de Thomas Jefferson, intitulée *Jefferson et son temps*, Dumas Malone raconte comment Jefferson a dirigé la première réunion tenue pour décider de l'organisation de la future Université de Virginie. L'université était au départ une idée de Jefferson, mais (comme c'est le cas partout, pas uniquement dans les projets libres)

1. Pour un autre point de vue je vous conseille la lecture de la thèse de doctorat de Martin Michlmayr : « Quality Improvement in Volunteer Free and Open Source Software Projects : Exploring the Impact of Release Management » (http ://www.cyrius.com/publications/michlmayr-phd.html) [NdT : « Amélioration de la qualité dans les projets volontaires de logiciels libres et Open Source : l'impact de la gestion des sorties »]. Elle traite de la gestion des sorties non plus sur la base des fonctionnalités, mais sur une base temporelle dans les projets importants de logiciels libres. Michlmayr s'est aussi exprimé sur le sujet chez Google, et la vidéo est disponible sur Google Vidéo à l'adresse http ://video.google.com/videoplay ?docid=-5503858974016723264.

beaucoup d'autres personnes se sont rapidement jointes au projet, chacun avec ses propres intérêts et ses propres plans. Lorsqu'ils se sont réunis la première fois pour débroussailler le terrain, Jefferson s'est présenté avec des plans architecturaux méticuleusement préparés, un budget détaillé pour la construction et le fonctionnement, une proposition de cursus et les noms de professeurs qu'il voulait faire venir d'Europe. Personne d'autre dans l'assemblée, et de loin, n'était aussi bien préparé, le groupe a donc dû s'en remettre à la vision de Jefferson et finalement l'université a été conçue à quelques détails près selon ses plans. Il avait certainement prévu que le coût de la construction dépasserait largement le budget et que beaucoup de ses idées ne fonctionneraient finalement pas (pour diverses raisons). Son but était purement stratégique : se présenter devant l'assemblée avec quelque chose de si solide que les autres en seraient réduits au rôle de consultants et qu'ils ne pourraient plus proposer que de simples modifications. Il s'assurait ainsi que le projet se déroulerait comme il le souhaitait dans les grandes lignes et selon le planning qu'il avait imaginé.

Concernant les logiciels libres, il n'y a pas de grande assemblée générale mais plutôt une série de petites propositions faites principalement par le biais du système de suivi de bogues. Mais si vous avez déjà une certaine crédibilité au sein du projet, et que vous commencez à prévoir différentes fonctionnalités, améliorations et bogues pour des versions précises et en accord avec un plan général annoncé, les participants vous suivront sans problème. Une fois que vous avez établi les choses à peu près comme vous le désirez, les conversations à propos de la date de sortie seront moins problématiques.

Il est bien sûr crucial de ne jamais présenter une décision unilatérale comme définitive. Dans les commentaires associés à chaque attribution d'un problème à une version particulière, initiez des discussions ou des protestations et montrez-vous authentiquement ouvert à la persuasion quand c'est raisonnable. Ne dirigez pas simplement pour le plaisir de diriger : plus les autres personnes participent au planning de sortie (voir la section « Partager les tâches de gestion aussi bien que les tâches techniques » du chapitre 8), plus il vous sera facile de les convaincre du bien-fondé de vos priorités sur les points qui vous tiennent à cœur.

Pour éviter de cristalliser les tensions autour du planning, le projet peut aussi proposer un rythme soutenu de publication. Quand le temps entre deux sorties est long, l'importance d'une nouvelle version est décuplée dans l'esprit des développeurs et ils sont beaucoup plus affectés si leur code n'y est pas incorporé parce qu'ils savent qu'il va leur falloir attendre longtemps avant d'avoir une nouvelle chance. Selon la complexité du processus de publication et la nature de votre projet, un délai de trois à six mois est en général un bon écart entre deux sorties, bien que les lignes de maintenance puissent soutenir un rythme plus important pour les micro-versions si elles sont nécessaires.

Encadrer les volontaires

La réussite d'un projet ne tient pas uniquement à la création d'une atmosphère sympathique ou à son bon fonctionnement. Cela ne suffit pas à mettre tout le monde d'accord sur les besoins du projet ou encourager la collaboration. Il faut qu'une ou plusieurs personnes prennent en charge l'encadrement des volontaires. Encadrer les volontaires n'est peut-être pas un art technique au même titre que la programmation, mais c'est un art, dans le sens où cette discipline peut-être améliorée par l'étude et la mise en pratique.

Le but de ce chapitre n'est pas de vous apporter sur un plateau un ensemble de techniques précises et prêtes à l'emploi pour diriger les volontaires. Il s'appuie, peut-être plus encore que les autres chapitres, sur le projet Subversion comme cas d'étude. Ceci est dû au fait que je travaillais sur ce projet au moment de la rédaction du livre, j'avais donc un accès direct à la matière première. Mais c'est aussi parce que « charité bien ordonnée commence toujours par soi-même ». J'ai également été témoin, dans d'autres projets, des conséquences de la mise en application des recommandations qui suivent, ou des conséquences de leur inapplication le cas échéant. S'il s'avère politiquement correct de citer des exemples venant d'autres projets, je le ferai.

En parlant de politique, quitte à en discuter, autant le faire maintenant et regarder de plus près ce domaine tant décrié. De nombreux ingénieurs pensent que la politique est une chose réservée aux autres. « Je ne fais que défendre la meilleure marche à suivre pour le projet, mais elle soulève des objections pour des raisons politiques ». Je pense que ce dégoût de la politique (ou de ce qui est perçu comme tel) est particulièrement fort chez les ingénieurs, car ils adhèrent à l'idée que certaines solutions sont objectivement meilleures que d'autres. Ainsi, quand quelqu'un se laisse guider par des considérations externes (par exemple : la défense de sa propre position d'influence, l'abaissement de l'influence d'une autre personne, un retournement de veste manifeste ou encore le désir de ne blesser personne), d'autres dans

le projet pourraient en être gênés. Ce qui ne les empêche évidemment pas de se comporter de la même manière quand leurs propres intérêts vitaux sont en jeu.

Si pour vous « politique » est un gros mot, et que vous espérez qu'il ne vienne pas salir votre projet, vous pouvez tout laisser tomber dès maintenant. La politique entre inévitablement en compte lorsque des personnes doivent gérer ensemble une ressource partagée. Lors d'une prise de décision, il est tout à fait normal de prendre en considération les conséquences de cette décision sur votre influence future dans le projet. Après tout, si vous avez confiance en votre propre jugement et vos propres compétences, comme c'est le cas de la plupart des programmeurs, le risque d'une perte d'influence doit être considéré, dans un sens, comme une donnée technique. Un raisonnement similaire s'applique à d'autres comportements qui peuvent, de prime abord, sembler purement politiques. En fait le « purement politique » n'existe pas, c'est bien parce que les actions ont des répercussions dans le monde réel que les citoyens deviennent « politiquement conscients ». La politique n'est rien d'autre finalement que la reconnaissance des conséquences d'une décision et leurs prises en compte. Si une décision particulière mène à un résultat que les participants trouvent techniquement satisfaisant, mais qui modifie les relations au pouvoir et donne un sentiment de mise à l'écart à certains personnages clés, cette considération devient au moins aussi importante que le résultat technique. L'ignorer n'est pas un signe de noblesse d'esprit, mais plutôt un manque de clairvoyance.

Alors, en lisant les conseils qui suivent, quand vous travaillez sur votre propre projet, souvenez-vous que personne n'est au-dessus de la politique. S'en donner l'air n'est qu'une stratégie qui peut parfois être très utile, mais, dans tous les cas, jamais une réalité. La politique n'est que le fruit du désaccord entre certaines personnes, et les projets qui réussissent sont ceux qui développent des mécanismes politiques pour gérer les désaccords de manière constructive.

1. Tirer le meilleur des volontaires

Pourquoi les volontaires travaillent-ils sur des projets de logiciels libres ? [1]

Quand on le leur demande, beaucoup disent que c'est parce qu'ils veulent produire un bon logiciel, ou qu'ils veulent être personnellement impliqués dans la réparation des bogues importants pour eux. Mais en général, ce ne sont pas là les seules raisons. Après tout, pouvez-vous imaginer un volontaire s'accrochant à un projet même si personne ne le félicite pour son travail ni ne lui prête attention dans une discussion ? Bien sûr que non. Il est évident que les contributeurs participent à un projet de logiciel libre pour des raisons qui vont au-delà du désir de produire du bon code. Comprendre les motivations réelles des volontaires vous aidera à les attirer et les faire rester. Le désir de produire un bon logiciel peut faire

1. Cette question a été traitée en détail dans un papier signé par Karim Lakhani et Robert G. Wolf, intitulé « Why Hackers Do What They Do : Understanding Motivation and Effort in Free/Open Source Software Projects » [NdT : « Pourquoi les hackers font ce qu'ils font : comprendre les motivations et les efforts dans les projets de logiciels libres/Open Source »]. Les résultats sont intéressants. Vous pouvez le consulter à l'adresse : http ://mitpress.mit.edu/books/chapters/0262062461chap1.pdf.

partie de ces raisons, avec le défi et l'enrichissement personnel qu'apporte le travail sur des problèmes complexes. Mais les humains ont également un désir inné de travailler avec d'autres, de donner et recevoir du respect au travers d'activités coopératives. Les groupes, impliqués dans ces activités, doivent produire des normes de comportement, de sorte qu'une position s'acquiert et se maintient par des actions qui servent les objectifs du groupe.

Ces normes ne vont pas toujours s'imposer d'elles-mêmes. Par exemple dans certains projets, on peut avoir l'impression qu'une position s'acquiert en écrivant fréquemment et verbeusement ; les développeurs Open Source expérimentés savent sûrement de quoi je parle. Ils n'arrivent pas à cette conclusion par accident, ils y parviennent parce que lorsqu'ils produisent des arguments longs et compliqués, ils obtiennent du respect, peu importe si ça aide vraiment le projet. Vous trouverez dans les prochains paragraphes quelques techniques pour créer une atmosphère propice à des actions à la fois « politiques » et constructives.

Déléguer

Déléguer ne se résume pas à répartir la charge de travail, c'est aussi un outil politique et social. Envisagez les conséquences quand vous demandez à quelqu'un de faire quelque chose. La plus évidente est que, s'il accepte, il fera le travail et pas vous. Mais en même temps, il se rend compte que vous lui faites confiance pour mener cette tâche à bien. De plus, si vous en avez fait la demande dans un forum public, il sait que le reste du groupe est averti de la confiance accordée. Mais il ne faut pas qu'il se sente obligé d'accepter, vous devez formuler la demande de manière à lui laisser la possibilité de décliner avec élégance l'offre s'il ne veut pas vraiment faire le travail. Si la tâche requiert une coordination avec d'autres membres du projet, vous lui offrez effectivement de s'impliquer plus, de créer des liens qui n'auraient autrement jamais été forgés, et peut-être de devenir une figure d'autorité dans une sous-partie du projet. L'implication supplémentaire peut être intimidante comme elle peut être encourageante en raison d'un plus grand sentiment d'engagement.

Pour toutes ces raisons, il vaut mieux, parfois, demander à quelqu'un d'autre de faire quelque chose, même si vous savez que vous pourriez le faire plus rapidement ou mieux vous-même. Bien sûr, certains impératifs d'efficacité vous y amèneront : si vous avez des tâches plus importantes à accomplir, peut-être que le faire vous-même serait illogique. Mais, même quand cette considération ne s'applique pas, vous pouvez choisir de déléguer le travail à quelqu'un d'autre parce que vous cherchez à l'impliquer davantage sur le long terme, même si cela signifie que vous devrez passer plus de temps à le surveiller au départ. L'inverse est également vrai, si vous vous proposez de faire de temps en temps le travail que quelqu'un d'autre ne veut pas ou n'a pas le temps de faire, vous gagnerez sa bonne volonté et son respect. Déléguer et échanger ne servent pas seulement à mener à bien des tâches ponctuelles, mais aussi à impliquer les volontaires dans le projet.

Distinguer clairement requête et assignation

Parfois, il est juste d'attendre qu'une personne accepte une tâche particulière. Par exemple, si quelqu'un est à l'origine d'un bogue ou qu'il produit des lignes de code ne correspondant manifestement pas aux standards du projet, il suffit de signaler le problème en montrant que

vous attendez qu'il s'en occupe. Mais dans d'autres situations votre demande n'aura pas forcément cette légitimité naturelle. La personne est libre de répondre favorablement à votre demande ou non. La bonne volonté d'un bénévole n'est pas chose acquise, et vous devez bien faire la différence entre ces deux situations pour adapter votre requête en conséquence.

Une personne à qui vous demandez de faire quelque chose, en expliquant clairement que vous pensez que c'est de sa responsabilité alors qu'elle ne partage pas ce sentiment, ne sera certainement pas bien disposée à le faire. Ceci rend la répartition des tâches très complexe. Dans un projet, les participants savent, en général, qui est expert dans chaque domaine, donc, quand un bogue est signalé, tout le monde pense à un ou deux correcteurs susceptibles de le réparer rapidement. Cependant, assigner cette mission à l'une de ces personnes, sans avoir son accord, pourrait la mettre dans une position inconfortable. Elle peut le ressentir à la fois comme une reconnaissance de son expertise, mais aussi comme si son savoir-faire était récompensé par une charge supplémentaire. Après tout, c'est en réparant des bogues que l'on acquiert de l'expérience, alors peut-être que quelqu'un d'autre pourrait s'occuper de ceux-ci ! (Si le système de suivi de bogues assigne automatiquement les problèmes à certains en se basant sur les informations contenues dans le rapport de bogue, les attributions se font de manière plus douce puisque la désignation est un processus automatisé et qu'elle ne traduit pas les attentes d'autres individus).

Vous serez parfois amené à encourager celui qui peut résoudre le bogue le plus vite et l'inviter à le faire, même si idéalement il faudrait répartir la charge de travail. Puisque vous ne pouvez pas créer une exception dès que la nécessité se fait ressentir (« Serais-tu prêt à jeter un œil à ce bogue ? » « Oui ». « OK, je te donne alors ce travail ». « OK. »), vous devriez plutôt formuler la demande comme une requête, sans pression. Quasiment tous les outils de suivi des bogues permettent d'associer un commentaire à l'assignation d'une tâche. Dans ce commentaire, vous pourriez dire quelque chose comme ceci :

> *« Je te confie cette tâche, A. Nonyme, parce que tu es celui qui est le plus familier avec ce code. Tu es libre de refuser si tu n'as pas le temps de t'y pencher malgré tout (et fais-moi savoir si tu préfère ne plus recevoir ce genre de demande dans le futur) ».*

Vous envoyez ainsi une requête claire tout en laissant la possibilité à la personne concernée de la refuser. Le public ici n'est pas seulement la personne en question, c'est tout le monde : le groupe entier a la confirmation publique de son expertise, mais le message indique clairement qu'elle est libre d'accepter ou de refuser la responsabilité.

Le suivi après la délégation

Lorsque vous demandez à quelqu'un de faire quelque chose, souvenez-vous l'avoir fait, et suivez son travail quoi qu'il arrive. La plupart des requêtes sont formulées sur des forums publics et suivent à peu près le même modèle « Est-ce que tu peux t'occuper de X ? Fais-le nous savoir, c'est pas grave si tu ne peux pas, il faut juste qu'on sache ». Si vous recevez une réponse négative, vous revenez à votre point de départ et vous devrez tenter une autre stratégie pour régler le problème X. Si la réponse est positive, gardez un œil sur le progrès fait dans ce secteur, et commentez l'évolution du travail (n'importe qui travaille mieux en

sachant que le résultat sera jugé). S'il n'y a pas de réponse après quelques jours, répétez votre demande, ou écrivez que vous n'avez pas reçu de réponse et que vous êtes à la recherche de quelqu'un d'autre pour cette tâche. Ou bien faites-la vous-même, mais annoncez auparavant que vous n'avez pas reçu de réponse après votre première demande.

Le but n'est pas d'humilier la personne en constatant publiquement son absence de réponse. Vos remarques devraient être tournées de telle sorte qu'elles n'aient pas cet effet. Le but est simplement de montrer que vous savez où vous en êtes, et que vous prenez en compte les réponses reçues. Les gens seront plus prompts à dire oui la prochaine fois, car ils verront (même de manière inconsciente) que vous êtes en mesure de remarquer le travail fourni puisque vous avez noté le fait, beaucoup moins visible, que quelqu'un n'a pas donné de réponse.

Remarquer les centres d'intérêt des acteurs du projet

Il est certain que votre attention leur fera plaisir. En général, mieux vous connaissez et gardez en mémoire les facettes d'une personnalité, plus elle se sentira à l'aise, et plus elle voudra travailler au sein de votre groupe.

Il y avait par exemple une distinction marquée dans le projet Subversion entre ceux qui voulaient atteindre la version 1.0 définitive (ce qui s'est finalement réalisé), et ceux qui voulaient principalement ajouter de nouvelles fonctionnalités et travailler sur des problèmes intéressants mais qui ne se souciaient pas de la date de sortie de la version 1.0. Il n'y avait pas de bonne ou de mauvaise approche, il y avait simplement deux types de développeurs, les deux abattant beaucoup de travail pour le projet. Mais on a rapidement compris que l'excitation de la dernière ligne droite avant la sortie de la version 1.0 était partagée par tous. Les communications électroniques peuvent être très trompeuses, vous pouvez ressentir que vos intentions sont partagées, alors qu'en fait ce n'est vrai que pour les personnes avec qui vous étiez en contact, tandis que les autres ont des priorités complètement différentes.

C'est en connaissant les attentes vis-à-vis du projet que vous serez à même de formuler vos demandes efficacement. En montrant que vous comprenez les aspirations des participants, même sans faire de requête liée, vous les rassurerez car ils se diront alors qu'ils sont plus qu'un simple grain de sable au milieu d'un désert uniforme.

Félicitations et critiques

Félicitations et critiques ne sont pas antonymes, elles sont très semblables de bien des manières. Toutes les deux sont des formes d'attention, et sont plus efficaces quand elles sont précises plutôt que vagues. Les deux devraient être utilisées à des fins particulières. Les deux perdent de leur valeur en cas de surabondance, louez trop ou trop souvent et vos louanges s'en retrouveront dévaluées. Il en va de même pour les critiques, bien qu'en pratique elles soient généralement plus marquantes et du coup moins sensibles à la déflation.

Un aspect très particulier des environnements techniques est que les critiques détaillées et objectives sont souvent prises comme une sorte de compliment (comme on l'a vu dans la section nommée « Reconnaître l'insolence » dans le chapitre 6), car elles impliquent que le travail analysé est suffisamment important pour qu'on prenne le temps de s'y attarder.

Cependant, ces deux qualités (« détaillées et objectives ») doivent être remplies pour que ceci soit vérifié. Par exemple, si quelqu'un fait une modification brouillonne du code, il est inutile (et même dangereux) de simplement faire le commentaire : « c'était brouillon ». Le manque de soin est en fin de compte une caractéristique de la personne, pas de son travail, et il est important que vos réponses portent sur le travail. La critique ne vaut que si elle est constructive. Détaillez, avec tact et sans méchanceté, les maladresses dans la contribution. Si c'est déjà la troisième ou quatrième modification négligente consécutive de cette personne, il convient de dire, encore une fois sans énervement, à la fin de votre commentaire que cette habitude a été remarquée.

Si vous ne constatez pas d'amélioration malgré les critiques, c'est une voie sans issue. La seule solution est que le groupe retire à cette personne les responsabilités pour lesquelles elle n'a manifestement pas les compétences, en évitant au maximum de la blesser, référez-vous aux exemples de la section « Transitions » plus loin dans ce chapitre. Ceci arrive rarement cependant. Nous réagissons généralement bien bien aux critiques précises, détaillées et qui indiquent (même de manière implicite) une attente d'amélioration.

Les louanges ne blesseront évidemment personne, mais ça ne veut pas dire que vous devez en abuser pour autant. C'est avant tout un outil : avant de l'utiliser, réfléchissez à vos motivations. Utilisez les compliments à bon escient, et évitez de commencer à remercier tout le monde pour tout et n'importe quoi, et en particulier, pour ce qui constitue leur activité normale et attendue. Si vous mettez le doigt dans l'engrenage, où vous arrêterez-vous ? Devriez-vous féliciter chacun pour avoir accompli son travail habituel ? C'est risqué, car si vous en oubliez certains, ils se demanderont pourquoi. Il vaut bien mieux exprimer son appréciation et sa gratitude avec parcimonie, pour des efforts inhabituels ou non sollicités, avec l'intention d'encourager ce genre de pratiques. Lorsqu'un participant s'implique plus, ajustez vos attentes et vos félicitations en fonction. Des félicitations répétées pour des choses banales perdent leur sens. Cette personne devrait plutôt ressentir que son activité accrue est maintenant considérée comme normale et naturelle, et que seul un travail encore plus poussé sera remarqué.

Ça ne veut pas dire que les efforts de cette personne ne devraient pas être reconnus bien sûr. Mais souvenez-vous que si le projet est bien construit, tout ce que fait cette personne sera visible de toute façon, et le groupe saura (et la personne saura que le groupe sait) tout ce qu'elle réalise. La reconnaissance ne s'exprime d'ailleurs pas seulement par des félicitations. En mentionnant, en passant, dans un sujet lié que la personne a accompli beaucoup de travail dans ce domaine, vous lui attribuez un statut particulier, vous pouvez aussi la consulter publiquement pour des questions relatives au code. Mieux : vous pouvez, par la suite, utiliser plus ostensiblement ce qu'elle a fait afin qu'elle voie bien que les autres lui font suffisamment confiance pour s'appuyer sur les résultats de son travail. Tout ceci n'est pas nécessairement calculé. Celui ou celle qui contribue beaucoup et régulièrement au projet le remarquera et acquerra automatiquement une position influente. Vous n'aurez en général rien de particulier à faire en ce sens, à moins que vous ne sentiez, pour une raison ou pour une autre, qu'un participant est sous-estimé.

Éviter le marquage de territoire

Faites attention aux participants qui tentent de revendiquer le contrôle exclusif sur certains domaines du projet, et qui semblent se réserver tout le travail sur cette partie, jusqu'à reprendre agressivement la main sur une tâche que d'autres ont commencée. Un tel comportement peut sembler bénéfique à première vue. Après tout, si cette personne a décidé de prendre plus de responsabilités et d'accroître son activité dans un domaine donné, c'est tant mieux… mais destructeur à long terme. En voyant un gros panneau « Défense d'entrer », les autres volontaires passeront leur chemin. Le code bénéficiera de moins d'attention, il sera donc plus fragile parce que tout repose sur les épaules de cet unique développeur. Pire encore, l'esprit de collaboration et d'égalité dans le groupe est rompu. En théorie chaque développeur devrait être libre de faire ce qu'il veut quand il le veut. Bien sûr, en pratique ce n'est pas si simple : les experts dominent leur(s) domaine(s) de compétence, et les novices s'en remettent fréquemment à eux. Mais tout ceci est avant tout volontaire : l'autorité informelle s'obtient au mérite, et cela ne doit pas être un processus actif. Même si la personne est effectivement compétente dans le domaine où elle recherche plus d'influence, il est crucial que cette autorité reste informelle, qu'elle respecte le consensus du groupe, et qu'elle ne serve pas à verrouiller le domaine en question.

Si un travail est rejeté pour des raisons techniques, c'est une autre histoire. On parle ici de technique, pas de chasse gardée. Il se peut très bien qu'une personne fasse la majeure partie des inspections dans un domaine, et tout devrait bien se passer tant qu'elle n'empêche personne d'autre d'en faire autant.

Afin d'éviter tout ce qui ressemble de près ou de loin à du marquage de territoire, de nombreux projets ont pris l'initiative de ne plus citer le nom des auteurs ou des responsables dans les fichiers sources. J'approuve complètement cette pratique : nous faisons de même dans le projet Subversion, et c'est plus ou moins la politique officielle de l'Apache Software Foundation. Un membre de l'ASF, Sander Striker, l'explique ainsi :

> *« À l'Apache Software Foundation nous n'encourageons pas la signature du code source par les auteurs. Les raisons sont diverses, ce n'est pas uniquement par crainte des répercussions légales. Le développement collaboratif est avant tout un effort collectif : développer et faire évoluer un projet en tant que groupe. Il faut savoir montrer sa reconnaissance, c'est sûr, mais il faut le faire d'une manière ne permettant pas de fausses attributions, même implicites. Et comment savoir quand ajouter sa signature ? Est-ce que vous signez de votre nom quand vous modifiez un commentaire ? Quand vous ajoutez une amélioration d'une ligne ? Est-ce que vous retirez le nom de l'auteur quand vous devez revoir le code qui, finalement, est transformé à 95% ? Et que dire de ceux qui vont modifier ça et là des documents, juste assez pour atteindre un quota virtuel de travail afin d'ajouter leur signature et, ainsi, avoir leur nom partout ?*
>
> *Il y a de meilleurs moyens d'exprimer sa reconnaissance, et ce sont ceux-là que nous privilégions. D'un point de vue strictement technique, les signatures des auteurs ne sont pas nécessaires, si vous désirez savoir qui a écrit une partie précise du code, vous pouvez consulter le*

> *logiciel de gestion de versions pour le découvrir. Les signatures se périment également. Voulez-vous vraiment être contacté en privé à propos d'un morceau de code écrit cinq ans auparavant et que vous étiez content d'avoir oublié ? »*

Les fichiers du code source d'un logiciel représentent sa carte d'identité. Ils devraient mettre l'accent sur l'aspect collaboratif du développement, et ne devraient pas être divisés en plusieurs petits fiefs.

Pour certains, les signatures de développeurs ne devraient pas être éliminées, car c'est la manière la plus directe d'identifier ceux qui accomplissent le plus de travail. Cet argument pose deux problèmes. Primo, comment fixer un quota de travail à réaliser pour avoir le droit d'apposer sa signature ? Secundo, il ne fait pas la distinction entre mérite et autorité : l'ancienneté n'est pas suffisante pour se prévaloir de la « propriété » d'un domaine. Il est cependant difficile, voire impossible, d'éviter ce genre de méprise lorsque le nom d'individus apparaît en haut des fichiers source. Dans tous les cas, les informations relatives au mérite sont déjà inscrites dans les journaux de gestion de versions ou dans d'autres outils discrets comme les archives des listes de diffusion, ainsi on ne perd rien en retirant les signatures des fichiers source eux-mêmes [1].

Si votre projet décide de bannir les noms en tête des fichiers sources, assurez-vous de ne pas aller trop loin. Par exemple, de nombreux projets ont une zone « contributions » regroupant des outils et des scripts d'aide, et leurs auteurs sont souvent des personnes extérieures au projet. Ils ne sont donc pas entretenus par le projet, et il vaut mieux que le nom de leur auteur y figure. Cependant, si un outil de ce type est inspecté et amélioré par des membres du projet, et que vous décidez finalement de lui donner plus de visibilité, vous pouvez, en supposant que l'auteur approuve, retirer son nom afin que le code ressemble à toute autre ressource entretenue par la communauté. Si l'auteur est tatillon sur ce point, des compromis peuvent être trouvés, exemple :

```
#indexclean.py: Remove old data from
# a Scanley index
#
# Original Author: K. Maru<>
# Now Maintained By: The Scanley Project <>
# and K. Maru.
#
# ...
```

1. Je vous invite à jeter un œil à la discussion « having authors names in .py files » [NdT : conservez les noms des auteurs dans les fichiers . py] (à l'adresse http ://groups.google.com/group/sage-devel/browse_thread/thread/e207ce2206f0beee, vous y trouverez de bons contre-arguments, en particulier dans l'intervention de William Stein. Ce qui fait la différence ici, je crois, c'est que parmi les auteurs, nombreux sont issus d'un monde (le monde des mathématiques académiques) où la norme veut que l'on cite les auteurs directement à la source et qu'ils y attachent beaucoup d'importance. En de telles circonstances, il vaut peut-être mieux ajouter les noms des auteurs dans les fichiers sources, avec une description précise des contributions de chacun puisque, pour la majorité des participants, ce genre de reconnaissance est attendue.

```
#indexclean.py: Nettoie les anciennes données
# d'un index de Scanley
#
# Auteur original: K. Maru<>
# Maintenant géré par: The Scanley Project <>
# et K. Maru.
#
# ...
```

Mais il vaut mieux éviter ce genre de compromis si possible. Les auteurs sont en général très ouverts à la discussion, car ils aiment que leurs contributions soient reprises par un projet.

Il faut surtout retenir qu'il existe une continuité entre le cœur et la périphérie d'un projet. Les principaux fichiers du code source du logiciel font évidemment partie du cœur, et la communauté doit clairement affirmer qu'elle en assure le maintien. Cependant, toutes les ressources ne sont pas nécessairement développées et maintenues par le projet. C'est le cas, par exemple, d'outils d'aide ou de morceaux de la documentation qui peuvent être le fruit du travail d'une personne extérieure. Rien ne vous empêche de les intégrer au projet voire à la distribution, mais ils font partie de la périphérie du projet. Il n'est pas indispensable d'appliquer la même règle à tous les modules d'un projet, tant que le principe suivant est respecté : les ressources maintenues par la communauté ne peuvent être la chasse gardée d'aucun de ses membres.

La bonne dose d'automatisation

Les machines peuvent accomplir toutes sortes de tâches, profitez-en par une automatisation maximale. Par expérience, une machine effectuera une tâche dix fois plus efficacement qu'un humain. Ce gain peut atteindre un facteur vingt, ou plus encore, pour les opérations très fréquentes ou complexes.

Vous considérer comme « dirigeant du projet », plutôt que développeur au sein d'un groupe, pourrait vous être utile ici. Parfois, les développeurs sont incapables de prendre le recul nécessaire pour réaliser que tout le monde gaspille son temps à accomplir manuellement des fonctions automatisables. Même ceux qui s'en rendent compte peuvent ne pas prendre le temps de résoudre le problème. Chaque tâche prise individuellement ne semble pas être si fastidieuse, et personne ne prend donc jamais la peine d'y remédier. Mais si chaque petite tâche est répétée maintes fois par un développeur, multipliez-cela encore par le nombre de développeurs et l'automatisation s'impose alors comme une évidence.

Ici je prends le terme « automatisation » dans une acception large, pour désigner non seulement les actions répétées où une ou deux variables changent chaque fois, mais aussi tout type d'infrastructure technique au service des participants. L'automatisation minimale a été décrite dans le chapitre 3, mais chaque projet rencontre ses propres obstacles. Par exemple, un groupe travaillant sur la documentation pourrait vouloir un site Web affichant en permanence la dernière version des documents. Comme la documentation est souvent écrite dans un langage de mise en forme, il se peut qu'il y ait une étape de compilation,

souvent assez complexe, permettant la création de documents affichables ou téléchargeables. Mettre en place un site Web, où cette compilation est automatisée à chaque publication des modifications, peut être difficile et demander beaucoup de temps, mais cela en vaut la peine, même si ça vous prend un jour ou deux de réalisation. Une documentation constamment à jour sera un grand progrès pour votre projet, même si le fait qu'elle ne le soit pas vous semble n'être qu'un désagrément léger et ponctuel.

Non seulement votre équipe réalisera un gain de temps considérable, mais elle sera aussi à l'abri des erreurs humaines inévitables lors de l'accomplissement manuel de procédures complexes. Les ordinateurs ont bien été inventés pour réaliser des opérations déterminées en plusieurs étapes, offrant ainsi aux humains du temps pour les choses plus intéressantes.

Les tests automatisés

Si les tests automatisés sont indispensables au développement de tout type de logiciel, ils le sont particulièrement pour les projets Open Source. Les tests automatisés (notamment les tests de non-régression) donnent aux développeurs une plus grande liberté dans des domaines qui ne leur sont pas familiers, et donc encouragent l'esprit d'exploration. Parce que la détection manuelle des problèmes est si complexe (schématiquement, il faut deviner où pourrait se situer la faille, puis faire divers tests pour déterminer si on a vu juste ou non), que des moyens automatisés de détection font gagner beaucoup de temps au projet. Les développeurs savent qu'ils travaillent avec un filet de sécurité, et peuvent modifier sans crainte des pans entiers du code, ce qui facilite le maintien à long terme du projet.

Les tests de non-régression
« Test de non-régression » fait référence aux tests menés pour détecter la réapparition de bogues déjà réparés. Les tests de non-régression sont utilisés pour éviter que les modifications du code endommagent le logiciel de manière inattendue. Plus le logiciel croît et devient complexe, plus il y a de chances que ces modifications engendrent des effets secondaires inattendus. Une conception rigoureuse du logiciel vous épargnera quelques problèmes, mais vous ne pouvez pas les éviter complètement.
Par conséquent, de nombreux projets disposent d'une gamme de tests, d'un programme séparé qui demande au logiciel des actions connues pour provoquer des bogues particuliers. Si la suite de tests parvient à déclencher un bogue, on parle de régression, c'est à dire qu'on a rendu caduque la solution d'un bogue sans le vouloir (voir aussi la définition sur Wikipedia).

Les tests de non-régression ne sont pas la panacée. D'une part, ils fonctionnent bien pour les programmes en ligne de commande (les logiciels qui emploient une interface utilisateur graphique sont bien plus complexes à tester automatiquement), et d'autre part, la structure de la suite de tests peut être, elle-même, plutôt compliquée. Elle n'est pas forcément simple à appréhender, et il faut aussi la maintenir. Pour qu'elle soit vraiment utilisable, il faut sans cesse l'optimiser, même si ça peut prendre un temps considérable. S'il est facile d'ajouter de nouveaux tests, les développeurs le feront, et vous détecterez plus facilement les bogues. N'oubliez pas que ces efforts de simplification porteront leurs fruits sur toute la durée de vie du projet.

Une règle commune à beaucoup de projets est la suivante : « Ne détruisez pas l'architecture ! », ou en d'autres termes : ne faites pas de changement qui pourrait empêcher de compiler ou de lancer le programme. Quand cette mésaventure arrive à un développeur, il ne fait en général pas le malin, et se fait un peu railler par ses pairs. Lorsque le projet utilise aussi une gamme de tests de non-régression, il y a souvent un corollaire à cette première règle : ne changez rien qui pourrait faire échouer les tests. Les erreurs graves sont plus facilement détectables si toute une batterie de tests de non-régression vérifie l'intégrité du programme automatiquement le soir, et envoie les résultats à toute la liste des développeurs ou à une liste de diffusion dédiée : voilà un autre exemple d'automatisation qui vous sera grandement bénéfique.

La plupart des développeurs volontaires sont d'accord pour prendre un peu plus de temps pour écrire les tests de non-régression quand le système est suffisamment clair et facile à utiliser. Chacun comprend qu'il faut que les changements soient accompagnés de tests, d'autant plus que c'est aussi une bonne occasion de développer la collaboration : souvent deux développeurs se partageront le travail de réparation d'un bogue, l'un écrivant le correctif et l'autre écrivant le test. Comme ce dernier aura finalement plus de travail, et qu'en plus l'écriture du test apporte moins de satisfaction que la réparation du bogue lui-même, il faut impérativement que la suite de test ne rende pas les choses plus désagréables qu'elles ne le sont déjà.

Certains projets poussent encore plus loin, et exigent que chaque correctif ou nouvelle fonction soit accompagné d'un nouveau test. Pour en évaluer la pertinence il faut prendre en compte plusieurs éléments : la nature du logiciel, la composition de l'équipe de développement ainsi que la facilité à écrire ces tests. Le projet CVS a pendant longtemps appliqué cette règle. En théorie, c'est une bonne règle puisque CVS est un logiciel de gestion de versions, il est par conséquent exclu de prendre des risques avec les données des utilisateurs. Le problème dans la pratique est que le test de non-régression de CVS est un énorme script shell fait d'un bloc (ironiquement nommé sanity.sh [NdT : santé mentale]), difficile à lire, modifier ou compléter. La difficulté d'ajouter de nouveaux tests et le besoin d'en associer à chaque correctif font que CVS n'encourage pas du tout l'écriture de correctifs. Quand je participais au projet CVS, j'ai vu des développeurs abandonner leur correctif lorsqu'ils apprenaient qu'ils devaient y associer un nouveau test à ajouter à sanity.sh, même si leur correctif était déjà prêt.

Il est courant de passer plus de temps à écrire un nouveau test de non-régression qu'à corriger le bogue. Mais CVS avait porté ce phénomène à son paroxysme : certains pouvaient passer des heures à parfaire leurs tests sans qu'ils ne marchent pour autant car un script Bourne shell de 35 000 lignes est tout simplement ingérable. Même les développeurs expérimentés traînaient les pieds lorsqu'ils devaient ajouter un nouveau test.

C'est simplement notre incapacité à comprendre les bienfaits de l'automatisation qui nous a poussés dans cette impasse. Il faut reconnaître aussi qu'adopter un véritable cadre de tests, à créer nous-mêmes ou prêt à l'emploi, aurait demandé un effort conséquent [1]. Mais l'avoir négligé a coûté bien plus au projet au fil des ans. Combien de correctifs et de nouvelles

1. Remarquez que ce n'est pas la peine de convertir tous les tests existants pour le nouveau cadre de tests, les deux peuvent très bien cohabiter et vous pouvez convertir les anciens tests uniquement quand le besoin s'en fait sentir.

fonctionnalités n'ont pas été incorporés aujourd'hui dans CVS à cause de l'obstacle que représente une suite de tests mal conçue ? On ne peut pas déterminer le nombre exact, mais il est sûrement bien plus élevé que celui des correctifs ou des nouvelles fonctionnalités auxquels renonceraient les développeurs pour créer un nouveau système de test (ou en intégrer un prêt à l'emploi). Cette opération ne prendrait qu' un temps limité, alors que le handicap lié à l'utilisation de la suite de tests actuelle perdurera si rien n'est fait.

Une politique stricte vis-à-vis de l'écriture des tests n'est pas forcément mauvaise pour autant, pas plus que l'écriture de votre système de test sous forme d'un script Bourne shell. Tout dépend de sa conception et des tests à effectuer. En résumé, lorsque le système de test devient une obstruction sérieuse au développement, vous devez agir : cela vaut pour tous les procédés routiniers se transformant en obstacles.

Considérez tous les utilisateurs comme des volontaires potentiels

Chaque contact avec un utilisateur est une opportunité de recruter un nouveau volontaire. Lorsqu'un utilisateur prend le temps d'écrire sur des listes de diffusion du projet, ou de remplir un rapport de bogue, il se distingue, et montre un plus fort potentiel d'investissement que la majorité des utilisateurs (dont le projet n'entendra jamais parler). Vous devez répondre à ce premier signe : s'il a décrit un bogue, remerciez-le pour son rapport, et demandez-lui s'il voudrait le corriger lui-même. S'il a écrit pour dire qu'une question importante manque à la FAQ ou signaler une carence dans la documentation du programme, reconnaissez le problème (en supposant qu'il existe effectivement), et demandez-lui s'il serait intéressé par l'écriture des passages manquants. Évidemment, la plupart du temps les utilisateurs déclineront. Mais ça ne coûte rien de demander, et chaque fois que vous le faites, vous rappelez à tous les lecteurs du forum que chacun est invité à prendre part au projet.

Ne cherchez pas uniquement à recruter de nouveaux développeurs ou de nouveaux rédacteurs pour la documentation. Entraîner simplement les volontaires à écrire de bons rapports de bogues porte ses fruits sur le long terme si vous ne passez pas trop de temps sur chaque cas, et s'ils continuent à envoyer des rapports de bogue dans le futur, ce qui est plus probable dans la mesure où le premier rapport a reçu des critiques constructives. Une réaction constructive n'est pas forcément un correctif pour le bogue signalé, même si ce serait évidemment l'idéal. Les contributeurs veulent surtout une écoute, la correction de bogue ne venant qu'ensuite. Vous ne pourrez pas nécessairement assouvir le second désir immédiatement, mais vous pouvez satisfaire le premier (ou plutôt le projet tout entier peut le faire).

Logiquement, les développeurs ne devraient pas s'en prendre à ceux qui font preuve de bonne volonté et remplissent des rapports de bogue manquant de précision. C'est ma bête noire personnelle : je remarque que des développeurs le font tout le temps sur les listes de diffusion ouvertes et cela crée des dommages. Quelques malheureux débutants posteront des rapports inutiles :

> Salut, je n'arrive pas à faire marcher Scanley. À chaque fois que je le lance, il plante. Est-ce que quelqu'un d'autre rencontre ce problème ?

Un développeur, qui a déjà vu ce genre de rapport des milliers de fois, mais qui ne se met pas à la place du débutant répondra :

> Qu'est ce que qu'on est censé faire avec si peu d'informations ? Mince ! Donne-nous au moins quelques détails, ta version de Scanley, ton système d'exploitation et l'erreur.

Ce développeur n'est pas parvenu à concevoir le point de vue de l'utilisateur, et il n'a pas pris en considération les effets qu'une telle réaction peut avoir sur les personnes assistant à l'échange. Naturellement, un utilisateur qui n'a aucune expérience de programmation, n'ayant jamais rapporté de bogue, ne saura pas comment écrire un rapport de bogue. Comment bien gérer cette situation ? Éduquez-le ! Donnez-lui envie de revenir :

> *« Désolé que tu rencontres des problèmes. Nous avons besoin de plus d'informations pour comprendre ce qui t'arrive. S'il te plaît, indique-nous ta version de Scanley, ton système d'exploitation et le message d'erreur exact. Le mieux que tu puisses faire est de nous envoyer une copie de ce que tu as exactement tapé et le résultat obtenu. Réfère-toi à [comment reporter un bug [1]] pour plus de précisions. »*

En répondant ainsi il est bien plus probable que vous obteniez les informations nécessaires de la part de l'utilisateur, parce que le message est écrit de son point de vue. Premièrement, vous montrez de la sympathie : tu as eu un problème, nous comprenons ta douleur (ce n'est pas nécessaire dans tous les cas, ça dépend de la gravité du problème et de l'agacement montré par l'utilisateur). Deuxièmement, plutôt que de rabaisser la personne parce qu'elle ne sait pas comment rapporter un bogue, cette réponse lui indique comment faire, et fournit les détails nécessaires pour que cela soit vraiment utile, de nombreux utilisateurs par exemple ne réalisent pas que « montre-nous l'erreur » veut dire « montre-nous le message d'erreur exact, sans commettre d'omissions ou faire de raccourcis ». Dès le premier contact avec un nouvel utilisateur, vous devez être clair à ce sujet. Finalement cette réponse conclut par un lien vers des instructions plus précises et détaillées à propos des rapports de bogue. Si vous avez fait ce qu'il faut, l'utilisateur prendra souvent le temps de lire ce document, et se conformera aux directives qui s'y trouvent. Vous devez donc, évidemment, le mettre à disposition. Il devrait donner des instructions claires à propos du type d'informations dont votre équipe de développement a besoin pour étudier un rapport de bogue. Le mieux serait qu'il évolue avec le temps en fonction des erreurs que les utilisateurs continuent à faire.

Les instructions concernant les rapports de bogue du projet Subversion représentent un exemple assez standard dans ce domaine (voir Annexe D). Notez la conclusion, c'est une invitation à produire un correctif pour réparer le bogue. Une telle invitation ne vous apportera pas un meilleur rapport correctif/bogue, car la plupart des utilisateurs capables de réparer les bogues savent déjà qu'un correctif serait le bienvenu, inutile de le leur dire. Elle a pour but de rappeler à tous les lecteurs, en particulier aux nouveaux venus, que le projet dépend des contributions de volontaires. Dans un sens, les développeurs ne sont pas plus responsables de la réparation des bogues que la personne les ayant signalés. C'est un aspect important

1. http ://www.scanley.org/how_to_report_a_bug.html

avec lequel de nombreux volontaires ne sont pas familiers. Une fois qu'ils le réalisent, il y a de meilleures chances qu'ils aident à la création du correctif, à moins qu'ils ne participent au développement du code en fournissant une méthode de reproduction plus complète ou en proposant de tester les correctifs que les autres personnes soumettent. Il faut que tous les utilisateurs réalisent qu'il n'y a pas de barrière imperméable entre eux et l'équipe du projet, que tout est question de degré d'implication et non pas de personnalité.

S'il faut en général être sévère vis-à-vis des réponses agressives, vous pouvez faire une exception quand les utilisateurs se montrent grossiers. De temps à autre, certains enverront des rapports de bogue ou des plaintes qui, contenu informatif mis à part, sont carrément irrespectueux. Souvent ils alternent insultes et flatteries, comme dans ce message écrit pas un utilisateur sur une liste de diffusion de Subversion :

> *« Comment se fait-il qu'après 6 jours, il n'y ait toujours pas d'exécutable pour les plateformes Windows ? ! ? C'est toujours la même histoire et c'est plutôt frustrant. Pourquoi sa création n'est-elle pas automatisée afin qu'il soit disponible immédiatement ? ? Quand vous sortez une version « RC », je pense que le but est que les utilisateurs testent la version, mais vous ne fournissez aucun moyen de le faire. Pourquoi prendre la peine d'avoir une période de mise à l'épreuve si on ne peut pas tester ? ? »*

Les premières réponses à ce message plutôt provocateur furent étonnamment modérées : les intervenants ont mis en avant le fait que le projet avait pour règle de ne pas fournir d'exécutables officiels. Ils ont suggéré — mais pas tous avec la même mesure — que l'utilisateur devrait plutôt se porter volontaire pour les faire lui-même s'ils sont si importants à ses yeux. Que vous le croyez ou non, sa réponse commençait par ces lignes :

> *« Tout d'abord laissez-moi vous dire que je pense que Subversion est génial et je suis reconnaissant des efforts fournis par les personnes impliquées. »*

. . . et son message continuait par des admonestations contre le fait que le projet ne fournissait toujours pas d'exécutable, sans pour autant se porter volontaire pour y remédier. Environ 50 personnes lui sont alors tombées dessus, et je ne peux pas dire que cela m'ait vraiment dérangé. La politique de tolérance zéro, évoquée dans la section « Tuez la vulgarité dans l'œuf » du chapitre 2, est un passage qui s'applique à ceux auxquels le projet a (ou voudrait avoir) une relation durable. Mais quand quelqu'un montre clairement, dès le départ, qu'il causera plus de problèmes qu'autre chose, il n'y a aucune raison qu'il se sente bien accueilli.

Heureusement, ces situations restent rares, en particulier quand les projets s'attachent, dès le premier contact, à inciter de manière constructive et courtoise les utilisateurs à participer.

2. Partager les tâches

Partagez les tâches de management aussi bien que les tâches techniques pour la bonne marche du projet. À mesure que le projet progresse, vous serez amené à faire beaucoup plus

de gestion : gestion des volontaires et gestion du flux d'information. Il n'y a pas de raison pour que vous ne partagiez pas cette charge. Rassurez-vous, inutile d'instaurer une hiérarchie verticale pour autant. En pratique, elle ressemblera plus à l'organisation d'un réseau pair à pair qu'à une hiérarchie militaire.

Parfois, les rôles de gestion sont attribués formellement, d'autres fois c'est spontané. Dans le projet Subversion, nous avons un responsable correctifs, un responsable traductions, des responsables documentation, des responsables parutions (bien que ces postes restent officieux) et un responsable difficultés. Si certains rôles ont été créés, d'autres se sont créés d'eux-mêmes. Quand le projet se développe, il est normal que de nouveaux postes se dessinent. Nous allons maintenant les détailler plus précisément.

À mesure que vous lisez la description des rôles, vous remarquerez qu'aucun ne requiert le contrôle complet sur un domaine en question. Le responsable parutions n'empêche personne de faire des changements dans le système de contrôle de versions, le responsable FAQ ne s'impose pas comme la seule personne à pouvoir modifier la FAQ, etc. Il s'agit de trouver l'équilibre pour endosser ces responsabilités, sans exercer un contrôle exclusif. Une part importante du travail de tout responsable est de repérer ceux qui travaillent dans son domaine et de les former à faire les choses à sa manière afin que les efforts de chacun s'additionnent plutôt que d'entrer en conflit. Les responsables devraient documenter leur manière de travailler : si l'un d'eux venait à quitter le projet, on pourrait ainsi immédiatement lui succéder.

Il se peut qu'il y ait des conflits : une place convoitée par deux personnes ou plus. Il n'y a pas de manière simple et juste pour résoudre ce problème. Vous pouvez proposer à chaque volontaire d'établir un projet (une « candidature »), et que tous les participants votent pour élire le meilleur. Mais c'est fastidieux et cela peut devenir gênant. Il est plus judicieux, à mon avis, de demander aux candidats de régler cela entre eux. Ils y arriveront en général, et seront plus satisfaits du résultat que si la décision leur avait été imposée de l'extérieur.

Responsable correctifs

Dans un projet de logiciel libre qui reçoit de nombreux correctifs, les suivre tous et rester au fait de ce qui a été décidé pour chacun d'eux peut être un vrai cauchemar, particulièrement s'ils ne sont pas centralisés. La plupart des correctifs sont envoyés sur la liste de diffusion des développeurs, mais pas tous. Certains apparaîtront en premier sur le suivi de bogues ou sur un autre site Web. Et la vie d'un correctif est tout sauf un long fleuve tranquille.

Si, en inspectant le correctif, quelqu'un décèle un problème, il le renverra parfois à son auteur pour qu'il y remédie. Cela donne généralement naissance à un processus itératif, parfaitement visible sur la liste de diffusion, dans lequel l'auteur renvoie des versions revues du correctif jusqu'à ce que le relecteur ne trouve plus rien à critiquer. Il n'est pas toujours évident de déterminer quand ce processus a atteint son terme. Si le relecteur met la modification sur le site alors le cycle est clairement achevé. Mais s'il ne le fait pas, peut-être n'en a-t-il simplement pas eu le temps, ou n'a-t-il pas l'accès nécessaire pour entrer en contact avec un autre développeur pour que ce dernier le fasse.

La soumission d'un correctif peut aussi être à l'origine d'une discussion libre, pas forcément à propos du correctif lui-même, mais sur le concept sous-jacent, à savoir s'il est bon ou mauvais. Il peut, par exemple, corriger un bogue, mais le projet préfère régler ce bogue d'une

autre manière, au sein de la résolution d'un problème plus général. Le fond de la discussion n'est pas prévisible par avance, et c'est le correctif qui stimule la découverte.

À l'occasion, un correctif proposé ne soulève aucune réaction. C'est, en général, parce que les développeurs manquent de temps à ce moment précis pour inspecter le correctif, et chacun espère qu'une autre personne le fera. Comme cette période n'a pas de limites déterminées, chacun attend que les choses soient prises en main, et d'autres tâches importantes arrivant sans cesse, un correctif peut facilement être ignoré sans que personne ne l'ait vraiment voulu. Non seulement le projet peut passer à côté d'un correctif utile, mais l'auteur qui a fourni cet effort est découragé. Le projet peut alors sembler un peu déconnecté, en particulier aux yeux de ceux qui envisagent d'écrire des correctifs, et c'est bien là le plus grave.

Le responsable correctifs doit donc s'assurer qu'aucun correctif ne « passe au travers des mailles du filet ». Il lui faut pour cela suivre chaque correctif jusqu'à ce qu'il n'évolue plus. Le responsable correctifs surveille tous les sujets des listes de diffusion liés à la publication de correctifs. Si le sujet aboutit à la sortie du correctif, alors il n'a rien à faire. S'il entre dans un processus itératif d'inspection/amélioration le responsable le suit. Si une fois le processus achevé le correctif n'est pas publié, il doit remplir un rapport de problème pointant vers la version finale et vers la liste de diffusion concernée, afin qu'il bénéficie toujours d'une certaine attention. Si le correctif répond à un problème existant, il ajoute une note au rapport avec les informations pertinentes plutôt que de commencer un nouveau rapport.

Quand un correctif ne suscite aucune réaction, le responsable correctifs patiente quelques jours, puis demande si quelqu'un compte l'inspecter. En général, sa question ne reste pas lettre morte : un développeur peut expliquer qu'il ne pense pas que le correctif doive être appliqué, en donnant ses raisons, ou qu'il pourrait le contrôler, auquel cas on en revient à l'un des mécanismes précédemment décrits. S'il n'y a toujours pas de réponse, le responsable peut ou non remplir un rapport pour ce correctif, à sa discrétion. Au moins, la personne l'ayant soumis recevra un retour.

Avoir un responsable correctifs a permis d'économiser beaucoup de temps et d'énergie à l'équipe de développement de Subversion. Sans une personne désignée pour prendre cette responsabilité, tous les développeurs se demanderaient sans cesse « Si je n'ai pas le temps de répondre à ce correctif maintenant, est-ce que je peux compter sur les autres pour le faire ? Devrais-je essayer de garder un œil dessus ? Mais si pour les mêmes raisons d'autres personnes font de même, on va alors dupliquer cet effort inutilement ». Le responsable correctifs évite ces « et si… ». Chaque développeur peut prendre la bonne décision, pour lui-même, au moment où il découvre le correctif. S'il décide de se charger de l'inspection, il peut le faire, le responsable correctifs agira en conséquence. S'il veut ignorer complètement le correctif, ce n'est pas grave, le responsable correctifs est là pour s'assurer qu'il ne tombera pas complètement dans l'oubli.

Ce rôle devrait rester formel car il revient alors à une personne qui devrait toujours être active. Dans Subversion nous avions envoyé nos annonces aux listes de diffusion des développeurs et des utilisateurs, plusieurs réponses sont arrivées, et nous avons sélectionné le plus rapide. Quand il a dû abdiquer (voir la section nommée « Transitions » plus loin dans ce chapitre), nous avons recommencé. Nous aurions pu attribuer ce rôle à plusieurs personnes,

mais nous avons choisi de ne pas le faire à cause de la transparence requise dans leurs conversations. Mais peut-être serait-il plus logique de se reposer sur plusieurs responsables pour traiter de très gros volumes de correctifs.

Responsable traductions

Pour un logiciel, « traduction » peut avoir deux significations très différentes. Il y a la traduction de la documentation en plusieurs langues d'une part et la traduction du logiciel d'autre part. Le logiciel traduit affiche les erreurs et les messages d'aide dans la langue choisie par l'utilisateur. Les deux tâches sont complexes mais, une fois la bonne infrastructure en place, elles sont bien dissociables du reste du développement. Ces tâches s'avérant semblables sous certains aspects, vous pouvez n'avoir qu'un seul responsable traductions pour les deux, mais il peut aussi être plus intéressant d'en avoir deux différents.

Dans le projet Subversion, nous n'avons qu'un responsable traductions qui s'occupe des deux. Il ne traduit pas vraiment lui-même, même s'il peut bien sûr y aider de temps en temps, mais en plus de sa langue maternelle, il devrait parler dix langues (douze si on compte les dialectes) pour pouvoir travailler sur chacune ! Son rôle est donc de gérer les équipes de traducteurs volontaires : il les aide à se coordonner et fait le lien entre les équipes et le reste du projet.

Le responsable traductions est utile car les traducteurs et les développeurs sont deux groupes bien distincts. Parfois, ils sont complètement étrangers au travail sur un dépôt de gestion de versions, ou même au travail au sein d'une équipe éparse de volontaires. Mais à d'autres égards, ce sont souvent les meilleurs volontaires : avec un savoir particulier dans un domaine particulier, ils ont entrevu un besoin et ont décidé de participer. En général, ils ont soif d'apprendre et sont enthousiastes au travail. Ils ont juste besoin de quelqu'un pour les guider. Le responsable traductions s'assure que la traduction se passe sans créer d'interférences inutiles avec le développement régulier. C'est aussi lui qui représente les traducteurs lorsque les développeurs doivent être informés d'une modification technique nécessaire pour aider l'effort de traduction.

Les compétences les plus importantes à avoir ne sont donc pas techniques mais diplomatiques. Dans Subversion, par exemple, nous demandons que pour chaque traduction, il y ait au moins deux personnes, afin que le texte puisse être relu. Quand un nouveau volontaire arrive en proposant de faire la traduction de Subversion, en malgache par exemple, le responsable traductions doit essayer de le mettre en relation avec quelqu'un ayant posté six mois auparavant avec le même désir de faire une traduction malgache, ou lui demander poliment de trouver un autre traducteur malgache pour qu'ils s'associent. Quand ce quota est atteint, le responsable traduction leur donne l'accès nécessaire, les informe des règles du projet (comment écrire un message dans le registre par exemple), puis garde un œil sur eux pour voir s'ils suivent bien ces règles.

Les échanges entre le responsable traductions et les développeurs, ou entre le responsable traductions et les équipes de traductions se font en général dans la langue maternelle du projet, c'est à dire, la langue depuis laquelle toutes les traductions sont réalisées. Pour la plupart des projets de logiciel libre, c'est l'anglais, mais cela importe peu du moment que

c'est en accord avec le projet (l'anglais est sûrement la langue la plus indiquée pour les projets désirant attirer une importante communauté de développeurs internationaux).

Au sein d'une équipe de traduction, les membres parleront en général entre eux dans leur langue commune, et l'une des tâches dévolue au responsable traduction est de mettre en place des listes de diffusions dédiées. Ainsi les traducteurs peuvent discuter plus librement, sans déranger les utilisateurs des listes principales du projet. La plupart d'entre eux ne comprendraient pas la langue de travail, de toute manière.

Internationalisation et Localisation
L'internationalisation (I18N) et la localisation (L10N) font toutes les deux partie du processus d'adaptation du programme pour fonctionner dans un environnement linguistique et culturel autre que celui dans lequel il a été écrit à l'origine. On pense souvent que ces termes sont interchangeables, mais en fait ce n'est pas vraiment la même chose. Voici l'explication que l'on peut trouver sur Wikipedia :
L'internationalisation d'un logiciel consiste à le préparer à la régionalisation [NdT : *localization* en anglais], c'est-à-dire à l'adaptation à des langues et des cultures différentes. Contrairement à la régionalisation, qui nécessite surtout des compétences en langues, l'internationalisation est un travail essentiellement technique, mené par des programmeurs.
Ainsi, modifier votre logiciel, pour prendre en charge l'encodage de texte Unicode sans perte, est de l'internationalisation puisque ça ne concerne pas une langue particulière, mais plutôt l'acceptation de textes dans n'importe quelle langue. D'un autre côté, faire que votre logiciel affiche tous les messages d'erreur en slovène lorsqu'il détecte un environnement slovène, est de la localisation. Donc, le rôle principal du responsable de traduction concerne la localisation et non l'internationalisation.

Responsable documentation

Maintenir la documentation du logiciel à jour est une tâche sans fin. Chaque nouvelle fonctionnalité ou amélioration ajoutée au code est potentiellement source de modification de la documentation. Aussi, lorsque la documentation du projet atteint un certain niveau d'achèvement, vous remarquerez que beaucoup de correctifs envoyés concernent la documentation et non pas le code. Simple effet de masse, il existe davantage de contributeurs compétents pour corriger la prose que pour élaborer du code : tous les utilisateurs sont des lecteurs, mais seulement un faible pourcentage d'entre eux sont des programmeurs.

Les correctifs pour la documentation sont en général plus simples à inspecter et à appliquer que les correctifs pour le code. Puisque leur nombre est élevé, mais le travail d'inspection faible, le ratio travail administratif sur travail productif est plus élevé pour les correctifs de documentation que pour le code. De plus, la plupart des correctifs demanderont sûrement quelques ajustements, de manière à assurer la cohérence de la documentation. Les correctifs bruts seront rarement applicables, ils se chevauchent ou sont interdépendants, et doivent donc être ajustés avant d'être validés.

Le traitement des correctifs de la documentation est exigeant. Le code source doit sans cesse être surveillé pour que la documentation reste à jour. Il est donc logique de dédier une

personne ou une petite équipe à cette tâche. Ils sauront exactement où et dans quelle mesure la documentation a pris du retard sur le logiciel, et pourront mettre au point des procédures avancées de traitement de grandes quantités de correctifs.

Bien sûr, cela n'empêche personne faisant partie du projet d'améliorer la documentation à la volée, en particulier de faire de petites corrections quand le temps le permet. Et le responsable correctifs (voir la section appelée « Responsable correctifs » plus tôt dans ce chapitre) peut suivre à la fois les correctifs pour le code et ceux pour la documentation, les archivant là ou les équipes de développement ou de documentation en ont besoin (si la quantité de correctifs dépasse une limite humainement tenable, séparer la tâche du responsable correctifs entre deux responsables, code et documentation, sera peut-être un bon premier pas). L'équipe de documentation doit maintenir la cohésion entre ceux qui se sentent responsables de la traduction afin qu'ils soient organisés, cohérents et toujours à jour. En pratique, cela passe par une connaissance intime de la documentation, une surveillance étroite du code source et des modifications apportées par les autres à la documentation, mais aussi par le suivi des correctifs de documentation entrants, et par l'utilisation de toutes ces informations pour faire tout ce qui est nécessaire en vue de maintenir la documentation en bonne santé.

Responsable difficultés

Le nombre de problèmes dans le suivi de bogues augmente proportionnellement au nombre d'utilisateurs. Par conséquent, même si vous corrigez les bogues et proposez un programme de plus en plus solide, vous devez toujours vous attendre à observer une augmentation du nombre de problèmes non traités, sans forcément qu'ils aient de liens entre eux. Non seulement vous verrez apparaître de plus en plus de doublons, mais les rapports mal rédigés seront également plus nombreux.

Les responsables difficultés aident à soulager ces problèmes en surveillant ce qui entre dans la base de données et en l'inspectant entièrement et régulièrement à la recherche de problèmes particuliers. Leur activité la plus courante est certainement d'arranger les rapports de problèmes entrants, soit parce que le rapporteur n'a pas rempli certains champs du formulaire correctement, soit parce qu'un rapport semblable est déjà archivé. Évidemment, le responsable difficultés sera vraiment à même de détecter les doublons s'il est familier avec la base de données de bogue du projet. À ce titre, si vous pouvez amener quelques personnes à se spécialiser dans la base de données de bogues, le nettoyage de celle-ci sera fait beaucoup plus efficacement que si chacun y apporte sa petite contribution. Cette tâche demande vraiment une connaissance approfondie de la base de données, et personne ne peut l'acquérir quand tout le groupe tente de la mener à bien dans le désordre général.

Les responsables difficultés peuvent aussi aider à attribuer les problèmes à des développeurs précis. Face à la masse de bogues pouvant être signalés, chacun ne suit pas la liste de diffusion avec la même attention. Cependant, si quelqu'un surveille tous les rapports entrants et connaît bien l'équipe de développement, il peut attirer l'attention des développeurs concernés sur des bogues particuliers. Des paramètres extérieurs sont à prendre en compte, comme le déroulement du développement ou les envies et le tempérament du destinataire. Il serait donc souhaitable que les responsables difficultés soient eux-mêmes développeurs.

Selon l'utilisation que fait votre projet du suivi de problèmes, les responsables difficultés peuvent modeler la base de données pour qu'elle reflète les priorités du projet. Dans Subversion, par exemple, nous essayons de donner une prévision en termes de versions pour la correction des bogues. Lorsqu'il est demandé « Quand sera réglé le problème X ? », nous pouvons ainsi répondre « Dans deux versions », même si nous ne donnons pas de date exacte. Les sorties sont représentées dans le suivi de problèmes comme des jalons à atteindre, un champ disponible dans IssueZilla [1]. La règle en vigueur est que chaque nouvelle version de Subversion apporte une nouvelle fonctionnalité principale ainsi qu'une liste de correctifs de bogues particuliers. Pour chaque version, nous nous fixons des objectifs et une liste de problèmes à corriger. La nouvelle fonctionnalité en fait partie puisqu'elle devient aussi un problème. Les utilisateurs peuvent alors consulter la base de données pour connaître les améliorations à venir dans les versions prévues. En général cependant, ces objectifs sont modifiés. À mesure que de nouveaux bogues sont rapportés, les priorités sont réorganisées et les problèmes doivent être déplacés d'un jalon à un autre, afin que chaque version reste faisable. Pour ceci, encore une fois, les personnes ayant une vue d'ensemble, du contenu de la base de données et des liens entre les problèmes, sont mieux placées pour prendre ces décisions.

Les responsables difficultés doivent aussi trier les problèmes obsolètes. Un bogue peut tout simplement être réglé au gré des modifications, même sans être spécifiquement ciblé. Et ce qui est un bogue pour une personne peut être une fonctionnalité pour une autre ; le projet peut simplement changer son point de vue. Débusquer les problèmes obsolètes n'est pas simple, une seule solution : passer en revue toute la base de données consciencieusement. Les révisions complètes deviennent plus fastidieuses avec le temps et l'accroissement du nombre de problèmes. Il existe un seuil au-delà duquel, pour conserver une base de données saine, votre seul recours sera de « diviser pour mieux régner » : classer les problèmes immédiatement quand ils sont rapportés, et les rediriger vers le bon développeur ou la bonne équipe. Le destinataire prend alors la responsabilité du problème jusqu'à son terme, le menant vers une résolution ou vers l'oubli selon la situation. Quand la base de données devient trop importante, le responsable difficultés se mue alors en coordinateur, il passe moins de temps à s'occuper du problème lui-même qu'à le rediriger vers la bonne personne.

Responsable FAQ

L'entretien de la FAQ est un problème étonnamment difficile. Contrairement à la majeure partie des autres documents du projet, dont le contenu est planifié à l'avance par leurs auteurs, une FAQ est un document très réactif (voir « Entretenir une FAQ »). Son amélioration est par nature imprévisible. Et parce qu'il est complété au fur et à mesure, le document peut facilement devenir incohérent et désorganisé, voire contenir des entrées en double ou très semblables. Même quand vous n'êtes pas confronté à ces problèmes, certaines questions sont souvent liées entre elles. Malheureusement, les liens ne sont pas toujours détectés, notamment si les entrées ont été faites à une année d'intervalle.

Le rôle du responsable FAQ est double. Premièrement, il assure la qualité globale de la FAQ par sa connaissance des sujets abordés, ainsi, quand des intervenants ajoutent de nouvelles questions déjà posées, ou liées à d'autres déjà posées, il peut réaliser les corrections

1. IssueZilla est le suivi de bogues que nous utilisons, c'est un dérivé de BugZilla.

nécessaires. Deuxièmement, il surveille les listes de diffusion du projet et les forums pour repérer les problèmes ou les questions récurrentes, puis il crée une nouvelle entrée à la FAQ si besoin est. Cette tâche n'a rien de simple : la personne doit être capable de suivre un sujet, reconnaître le cœur du problème soulevé, proposer une nouvelle entrée pour la FAQ, tenir compte des commentaires extérieurs (puisqu'il est impossible que le responsable FAQ soit expert dans tous les domaines couverts par la FAQ) et estimer quand le processus est achevé et que l'entrée peut être ajoutée.

Le responsable FAQ devient en général l'expert de la mise en page de la FAQ par défaut. De nombreux petits détails sont à prendre en compte dans la mise en forme de la FAQ (voir la section appelée « Traiter toutes les ressources comme des archives » dans le chapitre 6) ; quand une personne modifie la FAQ, elle oubliera souvent certains de ces détails. Ce n'est pas grave tant que le responsable FAQ passe derrière pour corriger ces erreurs.

Il existe différents logiciels libres aidant à l'entretien de la FAQ. Vous pouvez les utiliser tant qu'ils ne compromettent pas la qualité de la FAQ : faites attention à l'abus d'automatisation. Certains projets tentent de rendre l'entretien de la FAQ entièrement automatique, permettant à chacun de contribuer à la FAQ et d'en modifier les entrées, à la manière d'un wiki (voir la section appelée « Wikis » dans le chapitre 3). C'est flagrant pour ceux qui utilisent Faq-O-Matic [1], bien que les exemples que j'ai observés aient pu être dus à une surautomatisation, ce pour quoi Faq-O-Matic n'est pas prévu à l'origine. Quoi qu'il en soit, en optant pour les avantages qu'offrent une FAQ entièrement automatisée, vous devrez vous contenter d'un document de moindre qualité. Personne n'a une vue d'ensemble de la FAQ, personne ne peut remarquer les articles qui nécessitent une mise à jour ou qui sont devenus complètement obsolètes et personne ne surveille les liens entre les questions. Au final les utilisateurs trouveront rarement l'aide qu'ils étaient venus chercher dans la FAQ. Dans le pire des cas, elle les induira même en erreur. Utilisez tous les outils nécessaires à l'entretien de la FAQ du projet, mais ne tombez pas dans la facilité, la qualité de votre FAQ s'en ressentirait.

Vous pouvez lire l'article [2] de Sean Michael Kerner intitulé « La FAQ des FAQ », vous y trouverez des descriptions et des évaluations d'outils Open Source pour l'entretien d'une FAQ.

3. Transitions

De temps à autres, un volontaire occupant une place importante (par exemple responsable correctifs, responsable traductions, etc.) ne pourra plus remplir ses fonctions. Les raisons peuvent être diverses : charge de travail trop importante, mariage, nouveau bébé, nouvel employeur, etc.

Quand un volontaire se retrouve submergé ainsi, il ne le remarque pas forcément de suite. Ça se passe en douceur, il n'y a pas de moment précis où il prend conscience de son incapacité à assumer ses responsabilités. Il se fait discret pendant un certain temps, puis soudainement, vous verrez un accroissement d'activité quand, réalisant qu'il a négligé le projet

1. http ://faqomatic.sourceforge.net
2. http ://osdir.com/Article1722.phtml

pendant trop longtemps, il prend une nuit pour rattraper son retard. Puis vous n'entendrez plus parler de lui à nouveau pendant un certain temps, et vous verrez peut-être de nouveau un regain d'activité. Les démissions ne seront que rarement spontanées. Si c'est un volontaire, il travaille sur son temps libre. Démissionner revient alors à avouer que son temps libre a été définitivement réduit, ce qu'on a souvent du mal à reconnaître ou accepter.

C'est donc votre rôle, ou celui des autres membres du projet, de remarquer quand quelque chose se passe, ou plutôt ne se passe pas, et de demander au volontaire si tout va bien. Ne prenez pas un ton accusateur, c'est une demande amicale. Vous cherchez une information, pas à mettre l'autre personne mal à l'aise. De manière générale, la demande devrait être faite ouvertement devant le reste du projet, mais si des circonstances particulières vous poussent à le faire en privé, cela ne pose pas de problème. Il y a une bonne raison de faire la demande en public : si le volontaire répond qu'il ne sera plus capable d'accomplir le travail, un contexte sera déjà établi pour votre prochaine annonce publique : la recherche d'un nouveau volontaire pour le remplacer.

Il peut arriver qu'un volontaire ne remplisse pas le rôle qu'on attend de lui. Il ne le remarquera pas forcément ou peut refuser de l'admettre. Bien sûr, chacun peut connaître des difficultés au départ, surtout si la tâche est complexe. Cependant, si la personne n'est pas à la hauteur des responsabilités qui lui ont été confiées, malgré l'aide des autres participants, l'unique solution pour elle est de s'effacer et de donner sa chance à une autre. Si elle ne s'en rend pas compte d'elle-même, quelqu'un devra le lui dire. Je ne vois qu'une bonne manière de procéder, c'est un processus en plusieurs étapes où chacune est importante.

En premier lieu, vérifiez que vous n'êtes pas fou. Sondez en privé l'avis des autres membres du projet pour savoir s'ils partagent le même sentiment que vous par rapport au problème. Même si vous êtes déjà sûr de vous, cela permettra de faire savoir aux autres que vous envisagez de demander à cette personne d'abandonner son rôle. En général personne ne fera d'objection, les autres seront soulagés que vous vous chargiez de cette tâche embarrassante à leur place.

Ensuite, contactez le volontaire en question, en privé, et informez-le, gentiment mais de manière directe, du problème. Soyez précis et donnez autant d'exemples que possible. Assurez-vous de bien mettre en avant l'aide proposée par les autres mais que les problèmes ont persisté sans amélioration. Prenez tout votre temps pour rédiger cet e-mail mais, pour ce genre de message, si vous n'étayez pas vos observations, mieux vaut ne rien faire. Annoncez-lui que vous voudriez chercher un autre volontaire pour ce rôle, mais indiquez également qu'il existe bien d'autres manières d'apporter sa contribution au projet. Pour le moment ne dites pas encore que vous en avez parlé à d'autres, personne n'apprécie qu'on lui dise qu'il est le sujet d'une conspiration.

Les choses peuvent prendre différentes tournures ensuite. Le plus probable est que la personne sera d'accord avec vous, ou qu'au moins elle ne vous contredira pas, et cèdera volontairement sa place. Dans ce cas, proposez-lui de faire l'annonce elle-même et de vous laisser vous occuper du message pour lui trouver un remplaçant.

Il peut aussi être d'accord sur le fait qu'il y a eu des problèmes mais demandera un sursis (ou une nouvelle chance dans le cas d'une tâche ponctuelle comme responsable parutions). Suivez votre jugement, mais peu importe ce que vous faites, ne donnez pas votre accord simplement parce que vous pensez que la demande est raisonnable. Cela ne ferait que prolonger

l'agonie, pas la réduire. En général, les choses n'en sont pas arrivées là par hasard, la personne a déjà bénéficié de beaucoup d'opportunités, elle a déjà eu sa seconde chance. Nous avons eu le cas avec un responsable parutions qui ne faisait pas vraiment ce qu'on attendait de lui, voilà comment j'ai tourné mon e-mail :

> > Si tu veux me faire remplacer par quelqu'un > d'autre, je passerai la main sans problème. > J'aurais juste une demande, qui je l'espère > n'est pas déraisonnable. J'aimerais qu'on > m'accorde une sortie de plus pour pouvoir > prouver ma valeur. Je comprends totalement ton désir (je suis passé par là moi aussi !), mais dans le cas présent on ne devrait pas procéder ainsi. Ce n'est pas la première ou deuxième parution, c'est la sixième ou la septième… Et je sais qu'à chaque fois tu avais un sentiment d'inachevé (parce qu'on en a déjà parlé avant). Tu as donc déjà eu ta seconde chance. Il faut bien un dernier essai... Je pense que [cette dernière parution] devrait l'être.

Dans le pire des cas le volontaire peut montrer ouvertement son désaccord. Vous devrez alors accepter ce passage délicat : il faudra bien vous faire entendre coûte que coûte. Le moment sera venu de dire que vous en avez déjà parlé à d'autres (mais ne dévoilez pas leur identité sans leur permission puisque ces discussions sont censées rester confidentielles), et que, pour le bien du projet, vous ne pensez pas qu'il faille continuer ainsi. Insistez, mais ne devenez pas menaçant. Gardez en tête que pour la plupart des postes la transition devient effective au moment où une nouvelle personne commence le travail, pas au moment où son prédécesseur arrête de le faire. Si la controverse touche le rôle de responsable difficultés par exemple, vous ou n'importe quelle autre personne influente dans le projet peut solliciter un nouveau responsable difficultés à n'importe quel moment. La personne qui le faisait avant n'a pas vraiment besoin d'arrêter de faire son travail, tant qu'elle ne sabote pas (délibérément ou non les efforts du nouveau responsable.

Mais alors, plutôt que de demander à la personne de démissionner, pourquoi ne pas l'encadrer sous prétexte de lui apporter un peu d'aide ? Pourquoi ne pas utiliser deux responsables difficultés, correctifs ou autre ? L'alternative semble tentante, mais…

Bien qu'en théorie l'idée semble bonne, en général elle ne l'est pas. Ce qui fait que le rôle de manager fonctionne, ce qui le rend utile en fait, est la centralisation. Tout ce qui peut être fait de manière décentralisée l'est en général déjà. Pour que deux personnes accomplissent une tâche de gestion, il faut qu'elles communiquent, ce qui implique un délai supplémentaire, sans oublier les erreurs de communication potentielles (« Je pensais que tu avais pris le kit de premiers soins ! » « Moi ? Non, je pensais que tu l'avais pris ! »). Bien sûr, il y a des exceptions. Parfois deux personnes travaillent extrêmement bien ensemble, ou alors le rôle peut être naturellement partagé facilement entre plusieurs personnes. Mais ça ne fera pas une grande différence dans le cas d'une personne se démenant dans un rôle pour lequel elle n'est pas faite. Si, dès le départ elle avait pris la mesure exacte du problème, elle aurait demandé de l'aide beaucoup plus rapidement. Dans tous les cas, ce serait un manque de respect que de laisser quelqu'un continuer un travail auquel personne ne prêtera attention.

Lorsque vous demandez une démission, observez la discrétion de rigueur : il n'y a rien de pire que de se savoir observé et sous pression. J'ai fait cette erreur une fois, une erreur évidente quand j'y repense, j'ai adressé un courrier aux trois parties pour demander au responsable parutions de Subversion de démissionner au profit de deux autres volontaires. J'en avais déjà parlé en privé avec ces deux derniers, et je savais qu'ils étaient désireux d'endosser cette responsabilité. Alors j'ai pensé, naïvement et avec peu de tact, que je gagnerai du temps et m'éviterai des tracas en leur envoyant à tous un courrier pour commencer la transition. Je supposais que le responsable parutions était déjà au courant des problèmes et comprendrait ma démarche sans problème.

J'ai eu tort. Le responsable parutions de l'époque s'est senti insulté, à raison. C'est une chose de se voir demander de passer la main, c'en est une autre devant ceux qui prendront la main. Dès que j'ai compris ce qui l'avait insulté, je me suis excusé. Finalement, il a démissionné et continue, aujourd'hui encore, à être impliqué dans le projet. Mais son amour-propre en a été blessé, sans oublier que les nouveaux responsables prenaient leurs fonctions dans un contexte délicat.

4. *Committers*

Les *committers* sont vraiment un groupe à part dans les projets Open Source, et à ce titre, ils méritent une attention particulière. Les *committers* sont les seules exceptions dans un système qui, normalement, est aussi peu discriminatoire que possible. « Discrimination » n'est pas pris dans un sens péjoratif ici. Le rôle des *committers* est absolument essentiel, je ne pense pas qu'un projet puisse réussir sans eux. Le contrôle qualité demande un… contrôle. Beaucoup de personnes se sentent capables d'apporter des modifications au programme, mais peu le sont vraiment. Le projet ne peut pas reposer sur les seules opinions, il doit imposer des normes, et accorder un accès de *commit* à ceux qui les respectent [1]. D'un autre côté, créer une caste induit un rapport de force. Ce rapport de force doit être encadré afin de ne pas nuire au projet.

Dans la section « Qui vote ? » du chapitre 4, nous avons déjà parlé du mécanisme de recrutement de nouveaux *committers*. Ici, nous nous intéresserons aux critères d'évaluation des nouveaux *committers* potentiels et à la présentation de ce processus à la communauté entière.

1. Attention, l'accès de *commit* a une signification légèrement différente dans un système de gestion de versions décentralisé où chacun peut mettre en place un dépôt lié au projet, et s'accorder personnellement l'accès de *commit* à ce dépôt. Néanmoins, le concept d'accès de *commit* ne perd pas tout son sens : « accès de *commit* » devient synonyme d'« autorisation d'apporter des modifications au code qui seront incluses dans la prochaine version du logiciel ». Si pour les systèmes de gestion de versions centralisés, c'est un accès à la validation, dans les systèmes décentralisés, cela signifie voir ses modifications intégrées d'office à la distribution principale. Dans les deux cas, l'idée est la même ; la mise en œuvre de l'idée n'est pas si importante.

Choisir les *committers*

Dans le projet Subversion, notre premier critère de sélection est le Serment d'Hippocrate. Premier commandement : ne fais rien de mal. Notre critère principal n'est pas l'aptitude technique ni la connaissance du code, mais simplement le bon jugement dont fait preuve le *committer*. Un exemple de bon jugement ? Savoir simplement quoi ne pas incorporer par exemple. Une personne peut envoyer de petits correctifs, réparant des problèmes basiques du code ; mais si les correctifs s'appliquent proprement, s'ils ne contiennent pas de bogues et sont en accord avec le journal de messages du projet et les conventions d'écriture du code, et s'ils sont assez nombreux pour attester d'une certaine constance, alors un des *committers* pourra suggérer d'attribuer à ce développeur un accès de *commit*. Si au moins trois personnes disent oui et que personne ne s'y oppose, la proposition est acceptée. Nous ne savons pas s'il est capable de résoudre des problèmes complexes dans tous les domaines du code, mais ce n'est pas important : il a au moins montré qu'il est capable de juger de ses propres compétences. Les aptitudes techniques peuvent s'apprendre (et être enseignées), mais le jugement est en grande partie inné. C'est donc cette qualité que vous cherchez chez une personne à qui vous voulez donner l'accès de *commit*.

Quand on propose de donner l'accès de *commit*, et que cette suggestion est controversée, ce ne sont en général pas les aptitudes techniques de la personne qui sont mises en cause, mais plutôt son comportement sur les listes de diffusion et sur IRC. Parfois, un développeur fait preuve de grandes qualités techniques et d'une bonne éthique de travail, mais en même temps, il se montre systématiquement hargneux ou peu coopératif sur les forums publics. C'est un problème sérieux. S'il n'offre pas de signes d'amélioration, malgré nos allusions, alors nous ne lui offrirons pas les responsabilités de *committer*, même s'il est très doué. Dans un groupe de volontaires, les aptitudes sociales et la capacité à « bien jouer dans le bac à sable » sont aussi importantes que les aptitudes techniques brutes. Si votre choix de *committer* ne s'avère pas judicieux, ce n'est pas tant au niveau technique que les conséquences sont graves. La gestion de versions permet de revenir sur des mauvaises décisions. Par contre, le projet pourrait être forcé d'annuler l'accès de *commit* de cette personne, une chose qui n'est jamais plaisante et qui peut parfois être conflictuelle.

Beaucoup de projets insistent sur le fait que le *committer* potentiel doive faire preuve d'un certain niveau d'expertise technique et d'implication en proposant un nombre non négligeable de correctifs non triviaux ; ce qui veut dire que ces projets ne veulent pas seulement s'assurer que cette personne ne causera pas de dégâts, ils veulent savoir aussi si elle est susceptible d'apporter des améliorations au code. C'est compréhensible, mais attention, ne transformez pas l'accès de *commit* en une carte de membre d'un groupe fermé. La question à se poser est : « Qu'est ce qui sera le plus bénéfique au code ? » et non « Allons-nous abaisser le statut social associé au rôle de *commit* en acceptant cette personne ? ». Le but d'un accès de *commit* n'est pas de renforcer le statut d'une personne mais de permettre l'ajout de bons changements au code sans trop le perturber. Si sur 100 *committers* vous en avez 10 qui font de grands changements régulièrement et les 90 autres qui se contentent de corriger des erreurs de frappe et des bogues minimes quelques fois dans l'année, c'est toujours mieux que si vous n'aviez que 10 *committers*.

Révoquer un accès de *commit*

Quoi qu'il arrive, évitez de devoir révoquer un accès de *commit*. Chaque cas de figure est différent, mais vous y perdrez de l'énergie et un temps précieux, vous avez certainement mieux à faire.

Cependant, si vous devez vous y résoudre, cette discussion ne doit pas être publique. N'y participent que ceux pouvant décider par vote d'informer le principal intéressé de la situation. La personne ne devrait pas être engagée dans la discussion. On est loin de la transparence prônée précédemment mais, dans ce cas, c'est nécessaire. C'est d'abord une condition pour que chacun s'exprime librement. Ensuite, si le vote échoue, vous ne voulez pas forcément que la personne sache que son éviction a été considérée, vous voulez éviter les interrogations néfastes (« Qui était de mon côté ? » ; « Qui était contre moi ? »). En guise d'avertissement, le groupe pourrait décider de révéler à la personne la remise en cause de son accès de *commit*, mais c'est une décision de groupe. Personne ne devrait de sa propre initiative révéler l'existence d'une discussion et d'un scrutin censé être secret.

Révoquer l'accès de quelqu'un revient à rendre cette information publique (voir dans ce chapitre la section intitulée « Éviter le mystère »), essayez donc de faire preuve d'autant de tact que possible dans la manière dont vous le présentez au monde extérieur.

Accès de *commit* restreint

Certains projets ont différents niveaux d'accès de *commit*. Par exemple, certains participants peuvent jouir d'un accès de *commit* leur laissant toute liberté sur la documentation, mais n'ont pas le même accès au code. On retrouve en général ces accès limités dans les domaines de la documentation, de la traduction, des liens entre le code et d'autres langages de programmation, des instructions détaillées pour la création de paquets (par exemple : les Redhat RPM spec files, etc.) et les autres domaines pour lesquels une erreur n'aura pas d'incidence sur le code.

Puisque l'accès de *commit* ne relève pas que de la technique, mais aussi de la politique du projet (voir la section appelée « Qui vote ? » dans le chapitre 4), la question suivante se pose naturellement : à quels votes peuvent prendre part les *committers* restreints ? En fait tout dépend du découpage de votre projet en domaines de *commit* restreints. Dans Subversion, les choses sont simples : un *committer* restreint peut voter pour les questions qui touchent exclusivement à son domaine et pour rien d'autre. Ce qui est important c'est que nous avons mis en place un système de « vote conseil » (en gros, le *committer* écrit « +0 » ou « +1 (ne compte pas) » plutôt que simplement « +1 » sur son bulletin). Son vote ne compte pas formellement, mais ça n'est pas une raison pour le bâillonner complètement.

Les *committers* non-restreints ont accès à tous les domaines. De la même manière, ils peuvent aussi se prononcer sur tous les domaines. Ce sont eux, et eux seuls, qui peuvent voter pour l'admission de nouveaux *committers* (quelle que soit l'étendue de leur futur accès). En pratique pourtant, pour les *committers* restreints, cette tâche est souvent déléguée : n'importe quel *committer* non-restreint peut « parrainer » un nouveau *committer* restreint, et les *committers* restreints dans un domaine peuvent souvent choisir les nouveaux *committers*

de ce même domaine (c'est particulièrement pratique pour permettre un travail de traduction sans accroc).

Ce sont là des principes généraux, vous pouvez évidemment les adapter aux spécificités de votre projet, mais ils sont assez communs. Chaque *committer* vote dans la limite des privilèges qui lui sont attribués. Chaque *committer* devrait pouvoir exprimer son vote sur les questions qui sont de son ressort, mais pas sur les questions qui sont au-delà de son champ d'accès. Les votes sur les questions procédurales devraient par défaut incomber aux *committers* ayant tous les droits, à moins qu'il n'existe des raisons (décidées par les *committers* non-restreints) pour élargir le suffrage.

Concernant l'accès de *commit* restreint, il vaut souvent mieux que ce ne soit pas le logiciel de gestion de versions qui en fasse respecter les domaines d'accès. Je vous renvoie à la section appelée « Autorisation » dans le chapitre 3 pour en connaître les raisons.

Les *committers* en hibernation

Certains projets retirent automatiquement leur accès de *commit* à ceux qui ne valident plus de modifications pendant un certain temps (un an par exemple). Je n'en vois pas l'intérêt, je pense même que c'est contre-productif pour deux raisons.

D'abord parce que certains pourraient se sentir obligés de valider des modifications acceptables mais inutiles simplement pour éviter que leur accès de *commit* n'expire. Ensuite, parce que ça ne sert pas vraiment à grand-chose. Le statut de *committer* « consacre » le discernement dont fait preuve une personne. Ce n'est pas parce qu'elle s'éloigne du projet que son jugement se déprécie. Même si elle ne donne pas signe de vie pendant des années, qu'elle ne regarde pas le code et qu'elle ne suit pas les discussions autour du développement, au moment de sa réapparition, elle aura conscience du retard accumulé et agira en fonction. Vous aviez confiance en son jugement précédemment, alors pourquoi changer ? Si les diplômes obtenus au lycée n'ont pas de date limite de validité, alors les accès de *commit* ne devraient pas en avoir non plus.

Parfois un *committer* peut demander à être remplacé ou à être clairement marqué comme étant inactif sur la liste des *committers* (voir la section appelée « Éviter le mystère » ci-dessous pour plus de détails à propos de cette liste). Dans ce cas, le projet devrait évidemment satisfaire à la demande de cette personne.

Éviter le mystère

Même si les discussions pour accorder un nouvel accès de *commit* doivent être confidentielles, les règles et procédures n'ont pas besoin d'être secrètes. En fait, il vaut mieux les publier afin que l'on comprenne que les *committers* ne font pas partie d'une sorte de « carré VIP » mystérieux dont l'accès est interdit aux simples mortels, mais que chacun peut y aspirer en proposant de bons correctifs et en sachant se comporter en communauté. Dans le projet Subversion, ces informations apparaissent directement dans les directives développeurs puisque les personnes susceptibles d'être intéressées par les démarches menant à l'octroi d'un accès de *commit* sont celles qui envisagent de contribuer au projet.

En plus de publier les procédures, publiez aussi la liste des *committers*. On la trouve habituellement dans un fichier appelé « MAINTAINERS » ou « COMMITTERS » au premier niveau de l'arborescence du code source du projet. Elle devrait lister tous les *committers* non-restreints en premier, suivis par les différents domaines de *commit* restreints avec leurs membres respectifs. Tous les noms et adresses e-mail devraient y figurer, les adresses pouvant être masquées pour éviter le spam (voir la section nommée « Masquer les adresses dans les archives » dans le chapitre 3).

Puisque la distinction entre les *committers* non-restreints et restreints est bien marquée et définie, il est normal que la liste fasse cette distinction aussi. Mais les distinctions faites par la liste ne devraient pas aller au-delà. Des positions informelles s'établiront inévitablement au sein du projet, elles n'ont pas leur place ici. C'est une liste publique, pas un fichier de remerciements. Listez les *committers* soit par ordre alphabétique, soit par ordre d'arrivée.

5. Remerciements

Les remerciements sont la monnaie dans le monde des logiciels libres. Les participants peuvent invoquer toutes sortes de motivations mais je ne connais personne qui serait heureux de faire tout ce travail anonymement ou sous une autre identité. C'est tout à fait compréhensible : c'est votre réputation dans un projet qui détermine plus ou moins votre influence. De plus, vous pouvez très bien valoriser votre participation à un projet Open Source puisque désormais certains employeurs recherchent cette référence sur un CV. Et les raisons les plus évidentes ne sont pas nécessairement les plus fortes : les contributeurs veulent simplement qu'on les apprécie et recherchent instinctivement des signes montrant que leur travail est reconnu par d'autres. La promesse de remerciements est une motivation forte. Lorsque des contributions minimes sont reconnues, leurs auteurs sont incités à en faire plus.

Le projet conserve toujours une trace de chaque contribution, c'est d'ailleurs l'une des caractéristiques du développement collaboratif (voir chapitre 3). Dès que possible, utilisez les moyens existants pour exprimer votre gratitude de manière précise, et soyez explicite quant à la nature de la contribution. Dans un message de journal, n'écrivez pas simplement « Merci à A. Nonyme <> » si vous pouvez écrire à la place « Merci à A. Nonyme <> pour le rapport de bogue et les pistes de résolution ».

Dans le projet Subversion, nous avons une règle informelle, mais cohérente, pour remercier les personnes qui rapportent un bogue, soit dans le formulaire du problème s'il y en a un, soit dans le message de journal du *commit* qui règle le bogue. Un rapide tour d'horizon des journaux des *commits* de Subversion jusqu'au *commit* 14525 montre que dans 10% des *commits* figurent des remerciements nominatifs où apparaît l'adresse e-mail. Ces remerciements sont en général destinés aux personnes ayant rapporté ou analysé les bogues corrigés par ce *commit*. Remarquez qu'il ne s'agit pas du développeur qui a vraiment fait la validation, et dont le nom est déjà automatiquement enregistré par le logiciel de gestion de versions. Parmi les 80 et quelques *committers* (restreints ou non) que compte actuellement Subversion, 55 d'entre eux avaient déjà été remerciés (en général plus d'une fois) dans le journal de validation avant de devenir *committers* eux-mêmes. Cela ne prouve évidemment pas qu'être remerciés est un facteur de leur implication continue, mais au moins cela établit une atmosphère dans laquelle ils savent que leur contribution sera remarquée.

Ne mélangez pas remerciements de routine et remerciements exceptionnels. Quand vous débattrez d'un morceau de code particulier, ou d'une autre contribution apportée, vous pouvez reconnaître leur travail. Par exemple, dire « Les dernières modifications de Daniel au code delta nous permettent maintenant d'ajouter la fonctionnalité X » aide à savoir de quels changements vous parlez et, en même temps, montre votre appréciation pour le travail de Daniel. D'un autre côté, simplement dire « Merci à Daniel pour ses modifications au code delta » n'apporte rien. Aucune information nouvelle n'est transmise puisque le logiciel de gestion de versions et d'autres mécanismes ont déjà enregistré le fait qu'il a apporté des modifications. Éparpillez vos remerciements et ils perdent leur valeur (vous perdez aussi votre temps). Leur portée dépend de leur caractère exceptionnel par rapport aux commentaires positifs habituels affluant en permanence. Cela ne veut évidemment pas dire que vous ne devriez pas remercier les contributeurs. Assurez-vous simplement de le faire, sans surenchère. Ces quelques conseils devraient vous aider :

– Vous êtes libre d'exprimer toute votre gratitude dans une discussion éphémère. Par exemple, remercier quelqu'un en passant, durant une conversation IRC, pour la correction d'un bogue, idem pour une remarque dans un e-mail dédié principalement à d'autres sujets. Mais n'envoyez pas un e-mail juste pour remercier à moins qu'un acte vraiment exceptionnel n'ait été accompli. De la même manière, ne surchargez pas les pages Web du projet avec des remerciements. Une fois mis le doigt dans l'engrenage, vous ne saurez jamais où et comment vous en sortir. Ne mettez pas de remerciements dans les commentaires du code, il ne faut pas s'éloigner du but premier des commentaires qui est d'aider le lecteur à comprendre le code.

– Il est important de remercier les personnes les moins impliquées pour ce qu'elles ont fait. Même si cela peut sembler illogique, ça cadre avec l'idée que vous devez exprimer votre gratitude aux personnes qui dépassent vos attentes. Par conséquent, si vous remerciez trop les contributeurs réguliers pour un travail normalement attendu, vous montrez implicitement que vous attendez moins de leur part qu'ils ne le pensaient. C'est plutôt l'effet contraire que vous recherchez !

Il existe certaines exceptions à cette règle. Il est normal de remercier quelqu'un ayant fait ce qui était attendu de sa part si cela implique de consentir à des efforts sporadiques intenses sur des périodes données. L'exemple typique est le responsable sorties qui s'arme lourdement quand la sortie approche mais qui le reste du temps se met en hibernation (en hibernation en tant que responsable sorties, il peut très bien être actif en tant que développeur, mais c'est une autre chose).

– Comme pour les critiques et les remerciements, la gratitude devrait être précise. Ne remerciez pas les participants simplement parce qu'ils sont fantastiques, même s'ils le sont. Remerciez-les pour avoir fait quelque chose sortant de l'ordinaire, pour les bons points, précisez en quoi leur réalisation était extraordinaire.

Le projet devra trouver un équilibre entre l'obligation de vérifier que toutes les contributions individuelles sont reconnues et le besoin de préserver le projet en tant qu'effort collectif, et non pas un ensemble de faits individuels mis bout à bout. Gardez ceci à l'esprit, et essayez de faire pencher la balance en faveur du groupe, ainsi vous pourrez garder le contrôle.

6. Les fourches

Dans la section « Fourchabilité » du chapitre 4, nous avons vu en quoi la possibilité de fourcher influe sur la gestion du projet. Mais que se passe-t-il quand une fourche se concrétise réellement ? Comment devriez-vous gérer la situation et à quoi devriez-vous vous attendre ? Inversement, quand devriez-vous concrétiser une fourche ?

Là encore, tout dépend de la situation. Si certaines fourches sont dues à des différences de point de vue amicales mais irréconciliables sur la gestion du projet, bien plus nombreuses sont celles motivées par des désaccords techniques et des relations difficiles. Mais la distinction entre ces deux extrêmes n'est pas si simple, puisque les débats techniques ne sont pas dénués de sentiments personnels. À l'origine de toute fourche, par contre, on retrouve un groupe de développeurs, voire un développeur isolé, pour qui la collaboration avec certains devient plus problématique que constructive.

Dès qu'un projet fourche, il est difficile de distinguer la « vraie » fourche du projet « original ». On parlera de la fourche F issue du projet P, avec P poursuivant à l'identique le long d'un chemin naturel, alors que F s'aventure sur un nouveau territoire, mais ce constat simpliste est déjà teinté de subjectivité. C'est principalement un problème de perception : la majorité est source de vérité, si les gens pensent « blanc », c'est que c'est blanc. Ici, il n'y a pas de vérité objective à l'origine, pas de vérité qui saute aux yeux. Ce sont plutôt les perceptions qui forment cette vérité objective puisque, au final, le projet ou la fourche est une entité qui n'existe que dans l'esprit des gens de toute façon.

Si ceux qui sont à l'origine de la fourche pensent qu'ils font germer une nouvelle branche sur le projet principal, la question de la perception est résolue immédiatement et facilement. Tout le monde, développeurs et utilisateurs, verra la fourche comme un nouveau projet, avec un nouveau nom (peut-être basé sur l'ancien, mais facilement reconnaissable), un site Web distinct et une philosophie ou un but distinct. Cependant, les choses se compliquent quand les deux camps considèrent qu'ils sont les légataires de l'esprit original du projet, et qu'ils ont donc le droit de le poursuivre sous son nom d'origine. Si un organisme possède les droits sur le nom ou le contrôle légal du domaine ou des pages Web, la question est en général résolue légalement : en cas de litige, c'est cet organisme qui décidera de la légitimité de chaque groupe puisqu'il a toutes les cartes en main dans la guerre des relations publiques. Évidemment, les choses vont rarement aussi loin : les rapports de force sont connus, personne ne se lancera dans une bataille dont l'issue est connue d'avance et le bon sens s'applique.

Une fourche est, par nature, un vote de confiance. Donc, heureusement, dans la plupart des cas, il y a peu de doute quant à la position de chaque groupe, projet ou fourche. Si plus de la moitié des développeurs se prononcent en faveur du chemin tracé par la fourche, il n'y a, généralement, pas besoin de fourche, le projet peut simplement emprunter ce chemin à moins qu'il ne soit dirigé par un dictateur particulièrement borné. À l'opposé, si moins de la moitié des développeurs y sont favorables, la fourche est clairement la rébellion d'une minorité, et la courtoisie et le bon sens veulent qu'ils se considèrent alors comme la branche divergente plutôt que comme la ligne principale.

Gérer une fourche

Quelqu'un menace de fourcher votre projet ? Gardez votre calme et souvenez-vous de vos objectifs à long terme. L'existence d'une fourche n'est pas une menace en soi : la perte de développeurs et d'utilisateurs, voilà le vrai risque. Votre objectif alors n'est pas de bâillonner la fourche, mais d'en minimiser les effets néfastes. Vous pourrez être énervé, vous pourrez penser que la fourche n'est pas justifiée et qu'elle n'a aucune légitimité, mais exprimer cela en public ne peut que vous aliéner les développeurs indécis. Vous ne devriez pas forcer les développeurs à faire un choix entre les deux projets, et vous devriez être aussi coopératif que possible avec la fourche. En premier lieu, ne retirez pas à quelqu'un son accès de *commit* dans votre projet simplement parce qu'il a décidé de travailler sur la fourche. Ce n'est pas parce qu'il a commencé à travailler sur la fourche qu'il a soudainement perdu ses capacités à travailler sur le projet original, ceux qui étaient *committers* devraient le rester. Au-delà de ça, vous devriez exprimer votre souhait de voir la compatibilité maintenue entre la fourche et le projet, et dire que vous espérez que les développeurs feront des portages entre les deux chaque fois qu'ils le pourront. Si vous avez un accès direct au serveur hébergeant le projet, offrez publiquement à la fourche une aide matérielle pour ses débuts. Par exemple, proposez un historique complet et détaillé du dépôt de gestion de version, si les participants n'ont pas d'autre moyen de l'obtenir, afin qu'ils n'aient pas à débuter sans historique (cela n'est pas forcément nécessaire, cela dépend du logiciel de gestion de version). Demandez-leur s'ils ont besoin d'autre chose et fournissez-le si possible. Inclinez-vous, avec réserve, pour montrer que vous ne serez pas un obstacle, que vous voulez donner sa chance à la fourche et que son succès ne dépendra que de ses qualités.

Si vous devez faire tout cela, et le faire publiquement, ce n'est pas pour aider la fourche en fait, mais pour montrer aux développeurs que rester de votre côté est le pari le plus sûr en vous montrant aussi peu vindicatif que possible. En temps de guerre, il est parfois logique (stratégiquement parlant si ce n'est pas humainement) de forcer les autres à choisir leur camp, mais pour les logiciels libres, ce n'est presque jamais le cas. En fait, quand une fourche apparaît, certains développeurs ne se cachent pas pour travailler sur les deux projets et font de leur mieux pour entretenir leur compatibilité. Grâce à eux, les lignes de communications restent ouvertes après la séparation. Ils permettent à votre projet de bénéficier des nouvelles fonctionnalités intéressantes de la fourche (oui, la fourche peut proposer des choses attrayantes) et ils augmentent aussi la probabilité d'une fusion ultérieure.

Parfois une fourche gagne un tel succès que, même si elle était considérée comme une fourche au départ par ses instigateurs, elle devient la version préférée de tout le monde, et finalement supplante la version originale à la demande des utilisateurs. La fourche GCC/EGCS en est un exemple connu. Le GNU Compiler Collection (GCC, anciennement GNU C Compiler) est le compilateur de code exécutable Open Source le plus populaire et aussi l'un des compilateurs les plus portables au monde. En raison de désaccords entre les développeurs officiels de GCC et Cygnus Software [1], l'un des groupes de développement de GCC le plus actif, Cygnus créa une fourche de GCC appelée EGCS. Ils avaient décidé de ne pas entrer en compétition : les développeurs d'EGCS n'ont à aucun moment essayé de faire passer leur version de GCC comme étant la nouvelle version officielle. Ils se sont plutôt concentrés sur l'amélioration de EGCS en ajoutant des correctifs plus rapidement que

1. Maintenant partie intégrante de RedHat.

ne le faisaient les développeurs officiels de GCC. EGCS a gagné en popularité et finalement certains parmi les principaux distributeurs de systèmes d'exploitation ont décidé d'adopter EGCS comme compilateur par défaut à la place de GCC. Il était alors clair pour les développeurs de GCC que s'accrocher au nom « GCC », alors que tout le monde adoptait la fourche EGCS, n'empêcherait pas la migration et que la situation serait inutilement compliquée. Alors, GCC a adopté le code de base de EGCS : il n'y avait alors de nouveau plus qu'un seul GCC, mais largement amélioré grâce à la fourche.

Cet exemple montre que vous ne pouvez pas toujours voir une fourche comme une sorte d'adultère. Même si une fourche peut être douloureuse et indésirable, vous ne pouvez pas forcément prédire si elle réussira ou non Vous, et le reste du projet, devriez donc la surveiller et vous préparer, non seulement à en absorber des fonctionnalités et du code quand c'est possible, mais dans les cas les plus extrêmes, à la rejoindre si elle gagne le cœur du projet. Bien sûr, vous pourrez souvent prédire les chances de succès d'une fourche d'après les personnes qui s'y joignent. Si la fourche est initiée par les pires empoisonneurs et regroupe des développeurs mécontents qui n'apportaient, de toute façon, rien de constructif, c'est plutôt mieux pour votre projet. Cela vous évite de prendre un autre chemin et vous n'avez pas à vous soucier de l'importance de la fourche vis à vis du projet originel. Par contre, si vous voyez des développeurs influents et respectés soutenir la fourche, vous devriez vous demander pourquoi. Peut-être que le projet devenait trop restrictif. La meilleure décision est alors d'infléchir le projet principal en intégrant tout ou partie des actions envisagées par la fourche, c'est-à-dire éviter qu'elle ne se concrétise.

Amorcer une fourche

Tous les conseils donnés ici supposent que la fourche n'intervient qu'en dernier recours. Épuisez toutes les autres solutions avant de commencer une fourche. Amorcer une fourche est presque toujours synonyme de perte de développeurs, avec seulement la promesse vague d'en gagner de nouveaux plus tard. C'est aussi le début d'une nouvelle compétition : tous ceux qui s'apprêtent à télécharger le logiciel vont se demander : « Lequel devrais-je prendre, celui-ci ou cet autre-là ? » Que vous soyez le premier ou le deuxième, la situation est épineuse car une nouvelle question a été introduite. À en croire la théorie de l'évolution, les fourches sont bénéfiques pour l'écosystème du logiciel entier : le mieux adapté survivra, ce qui signifie que, finalement, tout le monde profite d'un meilleur logiciel. Cela peut être vrai du point de vue de l'écosystème, mais ce n'est pas vrai du point de vue d'un projet pris individuellement. La plupart des fourches ne réussissent pas et la majorité des projets ne sont pas enchantés d'être divisés.

Pour cette même raison, vous ne devriez pas utiliser l'argument de la fourche comme un argument extrémiste, « Faites ce que je veux ou je commencerai une fourche ! », parce que tout le monde sait que les fourches, n'arrivant pas à attirer de développeurs du projet originel, n'ont qu'une faible chance de survie. Tous les observateurs, pas seulement les développeurs, mais aussi les utilisateurs et ceux qui font les dépôts pour les systèmes d'exploitation, choisiront leur camp selon leurs propres critères. Vous devriez donc vous montrer réticent à l'idée de commencer une fourche, ainsi, si vous devez finalement vous y résoudre, vous pourrez affirmer avec crédibilité que c'était la dernière option.

Ne négligez aucun facteur lorsque vous évaluez les chances de succès de votre fourche. Par exemple, si beaucoup de développeurs sur un projet ont un employeur identique, même mécontents et secrètement favorables à une fourche, il y a peu de chance qu'ils l'annoncent ouvertement si leur employeur y est opposé. De nombreux programmeurs de logiciels libres aiment à penser que le fait d'appliquer une licence libre au code signifie qu'aucune entreprise ne peut en dominer le développement. C'est vrai que la licence est au final une garantie de liberté. Si certains veulent vraiment initier une branche, et ont les moyens de le faire, ils le peuvent. Mais en pratique, les équipes de développement de certains projets sont principalement financées par une entité, et vous ne pouvez pas ignorer purement et simplement ce soutien. Si cette entité est opposée à la fourche, il y a peu de chance que ses développeurs y prennent part, même s'ils le désirent secrètement.

Si, pour autant, vous êtes toujours certain de la nécessité d'une fourche, assurez-vous, en premier, de vos soutiens en privé, annoncez ensuite la naissance de la fourche de manière amicale. Même si vous êtes en colère ou déçu par les responsables en place, ne le dites pas dans le message. Exposez sans emportement les raisons qui vous ont poussé à prendre cette décision, et annoncez que cette manœuvre ne vise pas à nuire au projet. En supposant que vous la considériez bien comme une fourche (en opposition à une manœuvre d'urgence de préservation du projet originel), soulignez bien que vous utiliserez le code, mais pas le nom du projet et que vous choisissez un nom qui ne prêtera pas à confusion. Vous pouvez utiliser un nom qui contient ou fait référence au nom original tant qu'aucune confusion n'est possible. Évidemment, vous pouvez mettre en avant sur le page Web de la fourche le lien de parenté avec le projet initial, et même que vous espérez le supplanter. Essayez simplement de ne pas compliquer la vie des utilisateurs en les mêlant à une bataille identitaire.

Pour finir, vous pouvez commencer du bon pied en octroyant automatiquement l'accès de *commit* pour la fourche à tous ceux qui l'avaient déjà sur le projet originel, même à ceux qui montraient ouvertement leur désaccord sur la nécessité d'une fourche. Même s'ils n'utilisent jamais leur accès, votre message est clair : il y a des désaccords, mais pas d'ennemis, et les contributions de tous ceux qui en ont les compétences sont les bienvenues.

Licences, droits d'auteur et brevets

La licence choisie n'aura certainement que peu d'importance dans l'adoption de votre projet s'il s'agit une licence Open Source. Les utilisateurs adoptent, en général, un logiciel pour ses qualités et ses fonctionnalités, pas pour des détails de licence. Néanmoins, vous devez avoir une connaissance minimale des questions relatives aux licences, à la fois pour vous assurer que celle que vous choisissez pour votre projet est en accord avec ses objectifs, mais aussi pour être capable de débattre de son choix. N'oubliez pas, en revanche, que je ne suis pas avocat et qu'aucun conseil, dans ce chapitre, ne doit être vraiment considéré comme légal. En cas de besoin, vous devrez engager un avocat à moins de l'être vous-même.

1. Terminologie

Dans toute discussion sur les licences libres, ce qui saute aux yeux c'est la variété de mots différents pour désigner apparemment la même chose : free software (logiciel libre), Open Source (logiciel ouvert), FOSS, F/OSS et FLOSS. Commençons par les rassembler, les trier, puis les compléter par d'autres termes.

Logiciel libre — Logiciel livré avec son code source, qui peut être librement partagé et modifié. L'expression fut à l'origine inventée par Richard Stallman, qui l'a formalisée pour produire la Licence Publique Générale (GNU General Public License) tout en créant la Fondation pour le Logiciel Libre (Free Software Foundation [1]), afin d'en promouvoir le concept. Bien que le logiciel libre couvre le même champ que l' « Open Source », la FSF, entre autres, préfère le terme initial parce qu'il englobe

1. http ://www.fsf.org/

l'idée de liberté et le concept de logiciel librement redistribuable. Elle privilégie aussi l'idée d'un mouvement social à celui d'un mouvement technique. La FSF reconnaît que le terme « free » est ambigu, il peut signifier « gratuit » ou « libre» mais, réflexion faite, il semble que ce soit le meilleur terme, les autres possibilités en anglais ayant chacune leur propre ambiguïté (tout au long de ce livre, le terme « free » est utilisé au sens de « libre » et non au sens de « gratuit ») [1].

Logiciel Open Source — Autre nom pour désigner les logiciels libres, mais qui reflète une philosophique distincte : le terme « Open Source » a été inventé par l'Open Source Initiative [2] comme une alternative au logiciel libre afin de rendre ce type de logiciel plus attractif aux entreprises tout en le présentant comme une méthodologie de développement plutôt qu'un mouvement politique. Il était également question de tordre le cou à une autre idée préconçue selon laquelle ce qui est « free » (gratuit) est de mauvaise qualité. Bien que toute licence libre soit également Open Source, et vice versa (avec cependant quelques exceptions), la majorité des gens tendent à utiliser un terme plutôt qu'un autre et à le privilégier. En général, ceux qui préfèrent « logiciel libre » embrassent plutôt l'aspect moral ou philosophique sous-jacent, tandis que ceux qui préfèrent le terme « Open Source » ne donnent pas de connotation idéologique à leur choix ou, tout du moins, ne désirent pas mettre cet aspect en avant. Voir la section appelée « Libre contre Open Source » dans le chapitre 1 pour plus de détails sur l'histoire de ce schisme. Dans un article [3], la Free Software Foundation fait une excellente interprétation, totalement arbitraire, mais nuancée et juste, de ces deux termes. L'Open Source Initiative développe son point de vue dans les deux sections :« The Marketing Case » et « Why free software is too ambiguous ? » [4].

FOSS, F/OSS, FLOSS — On dit « jamais deux sans trois » et c'est aussi vrai pour les termes relatifs au logiciel libre. Le monde universitaire, cherchant peut-être précision et exhaustivité plus qu'élégance, semble s'être accordé sur FOSS, ou parfois F/OSS, pour « Free/Open Source Software ». Une autre variante rejoignant la liste est FLOSS, pour « Free/Libre Open Source Software » (libre est familier dans beaucoup de langues et ne souffre pas des ambiguïtés de « free » ; voir Wikipedia [5]). Tous ces termes désignent essentiellement la même chose : un logiciel pouvant être modifié et redistribué par chacun, parfois, mais pas toujours, avec l'exigence que l'œuvre dérivée soit librement redistribuable sous les mêmes termes.

Conforme au DFSG — Conforme au contrat social Debian [6]. C'est un test largement utilisé pour savoir si une licence donnée est vraiment Open Source (free, libre, etc.). La mission du projet Debian est de maintenir un système d'exploitation entièrement libre. Aussi, en l'installant, personne ne peut douter de son droit de modifier et de redistribuer tout ou partie de ce système. Le contrat social Debian regroupe l'en-

1. NdT : Notez qu'en français, on préfèrera l'expression « Logiciel Libre » pour éviter toute confusion, mais que l'expression anglaise *free software* est tout de même usitée.
2. http ://www.opensource.org/
3. http ://www.fsf.org/licensing/essays/free-software-for-freedom.html
4. Le document est accessible en ligne à cette adresse : http ://opensource.linux-mirror.org/advocacy/case_for_hackers.php
5. http ://en.wikipedia.org/wiki/FLOSS
6. http ://www.debian.org/social_contract#guidelines

semble des exigences que la licence d'un programme doit remplir pour qu'il soit inclus dans Debian. Le projet Debian ayant passé beaucoup de temps à réfléchir à la conception d'un tel test, la charte conçue a prouvé sa robustesse (voir Wikipedia [1]) et, à ma connaissance, ni la Free Software Foundation, ni l'Open Source Initiative n'ont formulé d'objection sérieuse à son encontre. Si vous savez qu'une licence est conforme à la charte, alors vous savez qu'elle garantit toutes les libertés importantes requises pour soutenir les dynamiques d'un projet Open Source (comme la possibilité de faire des fourches, même contre les souhaits de l'auteur original). Toutes les licences discutées dans ce chapitre sont conformes à la charte Debian.

Approuvé OSI — Approuvé par l'Open Source Initiative. Il s'agit d'un autre test largement usité pour savoir si une licence confère toutes les libertés nécessaires aux utilisateurs. La définition de l'OSI du logiciel libre est basée sur le contrat social Debian, et toute licence qui remplit l'une des définitions satisfait presque toujours l'autre. Il y eut quelques petites exceptions au cours des années qui ne remettaient en cause que des licences marginales et aucune de celles vraiment importantes présentées ici. À la différence du Projet Debian, l'OSI maintient une liste des licences approuvées [2]. Ainsi, « approuvé OSI » est sans équivoque : une licence est ou n'est pas dans la liste. La Free Software Foundation entretient aussi une liste de licences [3] et les classe en catégories, non seulement en fonction des libertés qu'elles offrent, mais aussi en fonction de leur compatibilité avec la Licence Publique Générale GNU. La compatibilité avec la GPL est un sujet important, couvert dans la section intitulée « La licence GPL et la compatibilité des licences », plus loin dans ce chapitre.

Propriétaire, source fermée — L'opposé de « libre » ou « Open Source ». Il renvoie aux logiciels distribués sous des termes traditionnels, basés sur un schéma de redevance où l'utilisateur paie chaque copie, ou sur d'autres termes suffisamment restrictifs pour empêcher de satisfaire les dynamiques de l'Open Source. Même un logiciel distribué gratuitement reste propriétaire si sa licence ne permet pas sa libre redistribution ou modification. Généralement, « propriétaire » et « fermé » sont synonymes. Cependant, « fermé » implique, en outre, que le code source ne peut être vu. À partir du moment où le code source ne peut pas être vu, ce qui est le cas de la plupart des logiciels propriétaires, la distinction n'a plus lieu d'être. Cependant, certains développeurs placent leurs logiciels sous une licence qui permet à tout un chacun d'avoir accès au code source. Ils appellent ceci « Open Source » ou « proche Open Source », etc., mais ce n'est qu'une tromperie. La visibilité du code source ne fait pas tout : ce sont les libertés associées qui comptent. Ainsi, la distinction entre « propriétaire » et « fermé » est rarement pertinente, et les deux termes peuvent être considérés comme synonymes. Quelquefois, on emploiera « commercial » comme synonyme de « propriétaire », sauf qu'à proprement parler les deux ne sont pas identiques. Un logiciel libre peut être un logiciel commercial. Après tout, si les acheteurs peuvent toujours en distribuer eux-mêmes des copies, rien n'empêche un logiciel libre d'être vendu. Son exploitation commerciale peut prendre plusieurs formes comme par exemple la vente de service, de support ou de certification. De nos jours, des entreprises très

1. http ://en.wikipedia.org/wiki/DFSG
2. http ://www.opensource.org/licenses/
3. http ://www.fsf.org/licensing/licenses/license-list.html

rentables se sont construites autour du logiciel libre. Il n'est donc clairement ni anti-commercial, ni anti-capitaliste. Cependant, le logiciel libre est anti-propriétaire par nature, et c'est le point essentiel sur lequel il diffère du modèle traditionnel où une copie équivaut à une licence.

Domaine public — Œuvre dont personne ne détient les droits. Concrètement, personne ne peut restreindre le droit de copier l'œuvre. « Domaine Public » n'est pas synonyme de « sans auteur ». Toute œuvre a un auteur. Que le ou les auteurs aient choisi de placer leurs travaux dans le Domaine Public ne change rien au fait qu'ils en sont les créateurs. Quand une œuvre est placée dans le Domaine Public, le code peut être incorporé dans un travail protégé par le droit d'auteur ; par la suite, une copie de cette œuvre sera placée sous le même droit d'auteur que l'ensemble de l'œuvre. Mais cela n'affecte en rien la disponibilité du travail originel qui reste toujours dans le Domaine Public. Ainsi, placer une œuvre dans le Domaine Public revient techniquement à la rendre libre, si l'on s'en réfère à la plupart des organisations certifiées logiciel libre. Cependant, il y a de bonnes raisons pour utiliser une licence plutôt que d'opter pour le Domaine Public : avec un logiciel libre, certaines restrictions peuvent être utiles, pas seulement au détenteur du droit d'auteur, mais également aux utilisateurs, comme la prochaine partie le précisera.

Copyleft — Une licence qui utilise le droit d'auteur légal pour atteindre le résultat opposé. Suivant votre interlocuteur, vous aurez deux définitions. Une plus laxiste qui comprend les libertés présentées dans ce chapitre, ou une plus rigoureuse qui ne permet pas seulement ces libertés mais les renforcera en les imposant aux travaux dérivés de l'œuvre. La Free Software Foundation utilise exclusivement la seconde définition. Une majorité de personnes adoptent la définition de la FSF, mais d'autres, y compris celles qui écrivent pour les médias à la mode, choisissent d'opter pour la première définition. Il n'est pas sûr que tous ceux qui utilisent ce terme soient conscients de l'importance de la distinction entre ces deux définitions. Le meilleur exemple, le plus strict et le plus proche de cette définition, est la Licence Publique Générale GNU, qui stipule que tout travail dérivé doit également être placé sous cette même licence ; voir la section « La GPL et la compatibilité de licence » plus loin dans ce chapitre pour plus de détails.

2. Les aspects des licences

Malgré la diversité des licences de logiciels libres, elles partagent toutes une base commune : n'importe qui peut modifier le code, n'importe qui peut le redistribuer, tant sous sa forme originale que sous une forme modifiée. De plus, les détenteurs des droits d'auteur et les auteurs ne fournissent aucune garantie (l'exonération de responsabilité est particulièrement importante, étant donné que l'on pourrait utiliser des versions modifiées sans même le savoir).

Les différences entre licences se résument à quelques détails récurrents :

Compatibilité avec les licences propriétaires — Quelques licences libres permettent aux programmes propriétaires d'incorporer du code couvert. Les termes de la licence

du programme propriétaire n'en sont pas affectés pour autant. Il reste fermé, mais il intègre une certaine proportion de code d'origine non-propriétaire. La Licence Apache, la Licence Consortium X, la licence de type BSD et la licence de type MIT sont autant d'exemples de licences compatibles avec une licence propriétaire.

Compatibilité avec d'autres licences libres — La plupart des licences libres sont compatibles entre elles. Du code, soumis à l'une de ces licences, peut être combiné avec du code régi par une autre. Le résultat peut être distribué sous l'une ou l'autre des licences sans violer les termes de la seconde. La Licence Publique Générale GNU fait notoirement exception. Elle exige que tout logiciel utilisant du code sous GPL soit, lui aussi, distribué sous licence GPL, et ce sans ajouter de restrictions autres que celles que la GPL requiert. La GPL est compatible avec quelques licences libres, mais pas avec toutes. Nous aborderons ce point plus en détail dans la section « La GPL et la compatibilité de licences » plus loin dans ce chapitre.

L'obligation de crédit — Certaines licences libres stipulent que toute utilisation du code couvert doit être accompagnée par une notice, dont l'emplacement et l'affichage sont d'habitude spécifiés, donnant crédit aux auteurs ou détenteurs des droits d'auteur sur le code. Ces licences sont souvent compatibles avec du code propriétaire : elles n'imposent pas nécessairement la liberté du travail dérivé, mais simplement la reconnaissance de la paternité du code libre.

La protection de marques déposées — Une variante d'obligation de crédit. Des licences protégeant les marques déposées spécifient que le nom du logiciel original (ou des détenteurs du droit d'auteur, ou de leur institution, etc.) ne peut être utilisé dans des travaux dérivés sans une permission écrite préalable. L'obligation de crédit impose de citer l'auteur tandis que la protection de marque déposée a l'effet inverse. Opposées dans leur application, elles expriment néanmoins toutes deux le même désir : la réputation de l'auteur du code original doit être préservée et transmise, mais sans être ternie par association (d'idées).

La protection de l'« intégrité artistique » — Quelques licences (la Licence Artistique, celle qui couvre la célèbre implémentation du langage de programmation Perl, et la licence de Donald Knuth TeX, par exemple) exigent que le code modifié soit clairement identifié comme tel. Elles permettent essentiellement les mêmes libertés que les autres licences libres, mais imposent certaines contraintes rendant l'intégrité du code original facile à vérifier. La popularité de ces licences ne s'étant pas étendue au-delà des programmes spécifiques pour lesquels elles furent écrites, elles ne seront pas traitées dans ce chapitre ; elles ne sont mentionnées ici que dans un souci d'exhaustivité.

Ces conditions, pour la plupart, ne s'excluent pas entre elles, et quelques licences en intègrent plusieurs. Elles ont ceci en commun que, si elles attribuent aux usagers le droit d'utiliser et/ou de redistribuer le code, en contrepartie elles imposent certaines contraintes. Par exemple, certains projets veulent que leur nom et leur réputation se répandent avec le code, ce qui justifie la contrainte supplémentaire d'une clause de marque déposée ou de crédit. Si celle-ci est trop pesante, ce fardeau peut potentiellement détourner les utilisateurs vers un paquet utilisant une licence moins exigeante.

3. La GPL et la compatibilité de licence

Les licences peuvent être regroupées en deux grandes catégories selon qu'elles sont ou non compatibles avec les licences propriétaires. On distingue donc la Licence Publique Générale GNU de toutes les autres licences. Parce que l'objectif premier des auteurs de la GPL est la promotion du logiciel libre, ils créèrent délibérément la licence de façon à ce qu'il soit impossible d'intégrer du code sous GPL au sein de programmes propriétaires. En particulier, parmi les conditions de la GPL (voir le texte intégral [1]), on retrouve ces deux dispositions :

1. Tout logiciel dérivé, tout logiciel contenant une quantité significative de code sous GPL doit être distribué lui-même sous GPL.
2. Aucune restriction additionnelle ne peut être placée sur la redistribution du logiciel original ou dérivé [2].

Avec ces conditions, la GPL parvient à fabriquer une liberté contagieuse. Une fois qu'un programme est licencié sous GPL, ses termes de redistribution sont viraux, ils se transmettent à tout autre contenu intégrant ce code. L'emploi de code sous GPL dans un programme fermé est donc strictement impossible. Cependant, ces mêmes clauses rendent aussi la GPL incompatible avec certaines autres licences libres. C'est typiquement le cas des licences imposant une exigence supplémentaire comme, par exemple, une clause de crédit. L'incompatibilité avec la GPL est flagrante puisque cette dernière stipule que « Vous ne pouvez pas imposer de nouvelles restrictions… ». Du point de vue de la Free Software Foundation, ces effets secondaires sont recherchés, ou en tout cas, ils ne sont pas regrettables. Non seulement la GPL assure la liberté de votre logiciel, mais elle en fait aussi un agent poussant les autres logiciels à devenir libres.

Quant à savoir s'il s'agit ou non d'une bonne façon de promouvoir le logiciel libre, c'est l'un des trolls les plus persistants sur Internet (voyez la section appelée « Éviter les trolls » dans le chapitre 6), et nous ne l'examinerons pas ici. Retenez surtout que la compatibilité de votre licence avec la GPL est un aspect à ne pas négliger. La GPL est de loin la licence Open Source la plus populaire. Elle est choisie par 68% des projets répertoriés sur Freshmeat [3]. La deuxième licence la plus populaire ne concerne que 6% des projets. Si vous voulez que votre code puisse se mélanger librement avec du code sous GPL — et il y a beaucoup de code sous GPL — alors vous devriez choisir une licence compatible avec la GPL. La plupart des licences Open Source compatibles avec la GPL le sont aussi avec les licences propriétaires : ainsi, un code sous une telle licence peut être incorporé à un programme sous GPL tout comme il peut être utilisé par un programme propriétaire. Bien sûr, les résultats de ces mélanges seront incompatibles entre eux, puisque l'un serait sous GPL tandis que l'autre serait sous une licence propriétaire. Mais ce souci s'applique uniquement aux travaux dérivés et non pas au code que vous distribuez directement.

1. http ://www.fsf.org/licensing/licenses/gpl.html
2. NdT : La formulation exacte est : « You may not impose any further restrictions on the recipients' exercise of the rights granted herein. »
3. http ://freshmeat.net/stats/#license

Heureusement, la Free Software Foundation entretient une liste [1] rassemblant les licences compatibles avec la GPL, et celles qui ne le sont pas. Toutes les licences abordées dans ce chapitre sont présentes dans cette liste, dans une catégorie ou l'autre.

4. Le choix d'une licence

Quand vous décidez de la licence à appliquer à votre projet, essayez de ne pas réinventer la roue, si possible. Ces deux arguments devraient vous convaincre d'opter pour une licence déjà existante :

- La familiarité. Si vous optez pour l'une des trois ou quatre licences les plus populaires, les utilisateurs ne se sentiront pas contraints de lire les dispositions légales avant d'utiliser le code, ils les connaissent déjà.
- La qualité. À moins que vous n'ayez une équipe de juristes à votre disposition, il y a peu de chances que vous parveniez à réaliser une licence juridiquement solide. Les licences mentionnées dans ce livre sont les produits de l'expérience et de mûres réflexions : si votre projet n'a pas vraiment de besoins spécifiques, c'est le meilleur choix que vous puissiez faire.

Pour appliquer l'une de ces licences à votre projet, rendez-vous à la section « Comment appliquer une licence à votre logiciel » dans le chapitre 2.

La licence MIT/X Window System

Si vous cherchez une licence qui permette l'adoption de votre code par le plus grand nombre de développeurs et de travaux dérivés possible, et que son éventuelle intégration à du code propriétaire ne vous dérange pas, choisissez la licence MIT/X Window System (appelée ainsi parce que c'est sous cette licence que l'Institut de Technologie du Massachusetts a déposé son code du système X Window). Cette licence véhicule un message simple : « Vous êtes libre d'utiliser ce code comme vous le souhaitez ». Elle est compatible avec la GNU GPL, elle est succincte, simple et facile à comprendre.

> Copyright (c) <year> <copyright holders> Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions :
>
> The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

1. http ://www.gnu.org/licenses/license-list.html

> THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

(Voir le texte : MIT License [1])

La Licence Publique Générale GNU

Si vous préférez que le code de votre projet ne soit pas utilisé par des programmes propriétaires ou que ce critère n'est pas décisif à vos yeux, choisissez la Licence Publique Générale GNU [2] (traduite en français [3]). La GNU GPL est probablement la plus utilisée des licences de logiciels libres dans le monde aujourd'hui ; cette reconnaissance mondiale est par ailleurs en soi l'un des principaux avantages de la GNU GPL.

Quand vous codez une bibliothèque (*library*) dont la finalité est d'être utilisée par d'autres programmes, veillez attentivement à ce que les restrictions imposées par la GNU GPL correspondent aux buts de votre projet. Dans certains cas, par exemple si vous essayez de remplacer une bibliothèque propriétaire concurrente qui fait la même chose, il serait plus stratégique d'appliquer à votre code une licence qui permette son incorporation dans des programmes propriétaires, même si vous ne le souhaitiez pas. La Free Software Foundation a conçu une alternative à la GNU GPL pour de telles pratiques : la GNU Library GPL, plus tard renommée en GNU Lesser GPL (plus communément désignée par l'acronyme GNU LGPL). La GNU LGPL est moins restrictive que la GNU GPL et peut être intégrée plus facilement à du code non-libre. Cependant, elle est relativement complexe et demande un effort de compréhension, aussi, plutôt que la GNU GPL, je vous recommande d'opter simplement pour une licence de type MIT/X dans ce cas.

La GNU Affero GPL : une version de la GNU GPL pour le code côté serveur
En 2007, la Free Software Foundation a publié une variante de la GPL appelée GNU Affero GPL*. Elle a été créée pour appliquer les mêmes clauses de partage introduites par la GPL aux entreprises, de plus en plus nombreuses, qui proposent des services distants, des logiciels qui fonctionnent sur leurs serveurs, des logiciels avec lesquels les utilisateurs n'interagissent qu'en ligne et qui ne sont donc jamais distribués aux utilisateurs sous forme de

1. http ://www.opensource.org/licenses/mit-license.php
2. http ://www.fsf.org/licensing/licenses/gpl.html
3. http ://fsffrance.org/gpl/gpl-fr.fr.html

code source ou d'exécutable.
Un grand nombre de ces services employaient des logiciels sous GPL, souvent après y avoir apporté des modifications, et pourtant ils n'avaient pas à partager ces modifications puisque le code n'était pas distribué.
La parade de la licence GNU AGPLv3 consiste donc simplement à ajouter à la GPL classique une clause pour les « Interactions à distance » indiquant que « … si vous modifiez le Programme, votre version modifiée doit offrir de manière visible à tous les utilisateurs interagissant à distance grâce à un réseau informatique… la possibilité de recevoir la Source Correspondante de votre version… gratuitement, grâce aux méthodes standard ou habituelles de copies de logiciel ». Les règles établies par la GPL sont ainsi applicables dans le nouveau monde de la fourniture de services d'applications. La Free Software Foundation recommande que la GNU AGPLv3 soit appliquée à tous les logiciels qui seront communément employés sur un réseau.
Attention, la AGPLv3 n'est pas directement compatible avec la GPLv2 (mais elle est évidemment compatible avec la GPLv3). De toute façon, la plupart des logiciels sous GPLv2 possèdent une clause « ou toute version ultérieure ». Vous pouvez donc simplement passer à la GPLv3 si et quand vous en avez besoin. Par contre, si vous devez mélanger votre code avec un programme strictement sous GPLv2 (sans la clause « ou toute version ultérieure ») la licence AGPLv3 n'est pas applicable.
Bien que sa rédaction soit un peu compliquée, la licence en elle-même est très simple : elle complète la GPLv3 par une clause supplémentaire pour les interactions à travers le réseau. L'article anglais de Wikipédia sur l'AGPLv3 est excellent [NdT : l'article français est nettement moins développé, mais a le mérite d'exister].

[* L'histoire de la licence GNU Affero GPL est un peu compliquée. La première version de la licence a été publiée à l'origine par Affero, Inc, qui l'avait écrite en prenant la GNU GPL version 2 comme base. On parlait alors de l'AGPL. Ensuite la Free Software Foundation a décidé d'adopter l'idée mais, entre-temps, la version 3 de leur GNU GPL était publiée, ils ont donc basé leur licence « Affero » dessus et l'ont baptisée licence « GNU AGPL ». L'ancienne licence Affero est maintenant un peu raillée. Si vous voulez des clauses proches de l'Affero, vous devriez utiliser la version GNU. Pour éviter toute ambiguïté appelez-la « AGPLv3 », « GNU AGPL » ou pour un maximum de clarté « GNU AGPLv3 ».]

La GPL, libre ou pas ?

En choisissant d'appliquer la licence GNU GPL à votre code, vous vous exposez, ainsi que votre projet, à de possibles débats sur le respect de la « liberté » de la GNU GPL. Étant donné qu'elle impose quelques restrictions sur l'utilisation du code, à savoir qu'il ne peut être redistribué sous aucune autre licence, elle n'est pas complètement libre, si ? Pour certains, l'existence de cette restriction signifie que la GNU GPL est « moins libre » que des licences plus permissives telles que la licence MIT/X-Style. Cet argument en amène souvent un autre : « plus libre » est mieux que « moins libre » (après tout, qui n'est pas en faveur de la liberté ?), il en découle alors que ces licences sont meilleures que la GNU GPL.

Ce débat est encore un autre « troll » populaire (voir la section appelée « Éviter les trolls », chapitre 6). Évitez de nourrir le troll, au moins dans les forums. N'essayez pas de prouver que la GNU GPL est « moins libre », « aussi libre » ou « plus libre » que les autres licences. Mettez plutôt en exergue les raisons spécifiques à votre projet qui ont motivé votre choix. Si votre choix a été guidé par la reconnaissance d'une licence, dites-le. Si l'un de vos critères était l'obligation d'utiliser une licence libre pour un travail dérivé, dites-le également, mais refusez de vous laisser embarquer dans un débat sur le caractère « plus » ou « moins » libre de la GNU GPL. La liberté est un vaste sujet, débattre de la terminologie ne vous mènera à rien.

Comme nous ne sommes pas sur une liste de diffusion, je profite de l'écriture de cet ouvrage pour vous confier que je n'ai jamais compris l'argument : « La GPL n'est pas libre ». La seule restriction que la GPL impose est de ne pas imposer de restrictions supplémentaires. Dire que cela restreint la liberté, pour moi, revient à dire que l'abolition de l'esclavage réduit la liberté parce qu'alors les gens ne peuvent plus posséder d'esclaves.

Oh ! et si vous finissez par être entraîné dans un tel débat, ne rendez pas les choses pires encore en faisant des analogies dangereuses.

Et à propos de la Licence BSD ?

De nombreux logiciels Open Source optent pour la licence BSD (ou une variante). La Licence BSD originale fut créée pour la Berkeley Software Distribution. Par sa libération, l'Université de Californie a dévoilé des parties importantes d'une implémentation Unix. Cette licence (dont le texte exact se trouve à la section 2.2.2 de la documentation [1]) était proche de l'esprit de la Licence MIT/X, sauf pour une clause :

> All advertising materials mentioning features or use of this software must display the following acknowledgement : This product includes software developed by the University of California, Lawrence Berkeley Laboratory.

ou en français :

> Tout support publicitaire faisant mention ou utilisant ce logiciel doit faire apparaître le message suivant : Ce produit inclut des logiciels développés par l'Université de Californie, Laboratoire Lawrence Berkeley

La présence de cette clause rendait non seulement la Licence BSD originale incompatible avec la GPL, mais elle établissait également un dangereux précédent : alors que d'autres organisations ajoutaient des clauses de publicité à leur logiciels libres, adaptant la mention « the University of California, Lawrence Berkeley Laboratory » à leur propre nom, les revendeurs de logiciels devaient afficher de plus en plus de messages. Heureusement, de nombreux

1. http ://www.xfree86.org/3.3.6/COPYRIGHT2.html#6

projets soumis à cette licence, constatant le problème, abandonnèrent tout simplement cette clause. En 1999, même l'Université de Californie en fit de même.

La Licence BSD révisée n'est donc que la licence BSD originale épurée de la clause de publicité. Cette histoire rend malgré tout ambiguë la mention « Licence BSD » : parle-t-on de la version originale ou de la version révisée ? C'est pourquoi je préfère la Licence MIT/X qui est en substance équivalente, et qui ne souffre pas de cette ambiguïté. Mais il peut y avoir une raison de préférer la Licence BSD dans sa forme révisée à la Licence MIT/X, car la Licence BSD inclut cette clause :

> Neither the name of the <ORGANIZATION> nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

ou en français :

> Ni le nom de <ORGANISATION>, ni les noms des participants ne peuvent être employés pour approuver ou promouvoir les produits dérivés de ce logiciel sans accord écrit préalable.

Sans cette clause, il est difficile de savoir si un bénéficiaire du logiciel aurait le droit ou pas d'utiliser le nom du propriétaire de la licence : elle enlève tout doute possible. Pour les organisations attachant une importance particulière au contrôle des droits d'auteur, la Licence BSD révisée serait donc légèrement mieux adaptée que la Licence MIT/X. En général cependant, une licence de droit d'auteur libre n'implique pas que les bénéficiaires aient un quelconque droit d'utilisation ou de masquage de vos marques, les lois sur le droit d'auteur et sur la protection des marques sont deux choses différentes.

Si vous désirez utiliser la Licence BSD révisée, vous en trouverez un modèle sur le site opensource.org [1].

5. Attribution des droits d'auteur et propriété

La gestion des droits d'auteur sur un code libre, fruit du travail de plusieurs personnes, est une question qu'il vous faudra résoudre. Trois options s'offrent à vous. La première consiste à purement et simplement ignorer le problème du droit d'auteur (je vous le déconseille). La deuxième est de demander un Accord de Licence du Participant (ALP [2]) à chaque personne travaillant sur le projet, accordant explicitement au projet le droit d'utiliser le code du participant. C'est en général suffisant pour la plupart des projets, le point positif étant que dans certaines juridictions ces ALP peuvent être envoyés par courrier électronique. La troisième possibilité est d'obtenir des participants l'attribution complète des droits d'auteur afin que le projet (c'est-à-dire une personne morale, généralement à but non lucratif) concentre tous les

1. http ://www.opensource.org/licenses/bsd-license.php
2. NdT : ou CLA en anglais, pour Contributor License Agreement.

droits d'auteur. C'est la voie la plus sûre légalement, mais aussi la plus contraignante pour les participants, seuls quelques projets l'empruntent [1].

Notez que même si la propriété intellectuelle est centralisée, le code [2] demeure libre car les licences Open Source ne donnent pas au détenteur des droits la possibilité de rendre rétroactivement propriétaires toutes les copies de ce code. Donc, même si le projet, en tant que personne morale, retourne soudainement sa veste, et décide de distribuer le code sous une licence plus restrictive, cela ne posera pas de problème à la communauté. Les autres développeurs n'auraient qu'à initier une fourche basée sur la dernière version libre du code, et continuer comme si de rien n'était. Cette possibilité existant, la plupart des participants coopèrent lorsqu'on leur demande de signer un ALP ou une attribution de droit d'auteur.

Ne rien faire

La plupart des projets ne demandent jamais aux participants de signer d'ALP ou d'attribution de droits d'auteur. Ils se contentent d'accepter les contributions. En demandant leur incorporation, les participants donnent implicitement leur accord.

En temps normal, c'est suffisant. Mais il est déjà arrivé qu'un participant décide d'intenter un procès pour violation de droit d'auteur, affirmant qu'il est le vrai propriétaire du code en question et qu'il n'avait jamais donné son accord pour qu'il soit distribué par le projet sous une licence Open Source. C'est, par exemple, l'origine du litige entre SCO et le projet Linux (voir Wikipedia [3] pour les détails). Ne détenant aucun document officiel attestant de la cession de leurs droits par les participants, le projet peut difficilement se défendre.

Accord de Licence du Participant

Les ALP offrent certainement le meilleur compromis entre sécurité et aspect pratique. Un ALP est typiquement un formulaire électronique qu'un développeur remplit et renvoie au projet. Dans de nombreuses juridictions, l'envoi par courrier électronique est jugé suffisant. Une signature électronique sécurisée peut être requise, mais pas toujours. Consultez un avocat pour savoir ce qui serait le mieux pour votre projet.

La plupart des projets utilisent deux ALP légèrement différents, un pour les personnes, et un pour les entreprises. Mais dans les deux cas, le langage de base est le même : le participant accorde au projet

> *« … un droit d'auteur perpétuel, dans le monde entier, non-exclusif, sans frais, libre de droit, irrévocable pour la reproduction, la préparation d'œuvres dérivées, pour l'exposition publique, la présentation publique, pour sous-licencier et distribuer [les] Contributions et toute œuvre dérivée. »*

1. Le transfert de droit d'auteur est sujet aux lois nationales, les licences créées pour les États-Unis pourraient ne pas être adaptées partout (par exemple, en Allemagne, il n'est apparemment pas possible de transférer son droit d'auteur).

2. Je parlerai de « code » à partir de maintenant pour parler indifféremment du code et de la documentation.

3. http ://en.wikipedia.org/wiki/SCO-Linux_controversies

Par mesure de sécurité vous devriez faire valider votre ALP par un avocat, mais en réunissant tous ces adjectifs, il ne devrait rien trouver à redire.

Lorsque vous demandez aux participants de signer un ALP, insistez bien sur le fait que vous ne demandez pas une attribution des droits d'auteur. En fait de nombreux ALP commencent par un rappel au lecteur :

> This is a license agreement only ; it does not transfer copyright ownership and does not change your rights to use your own Contributions for any other purpose.

ou en français :

> Ceci n'est qu'un accord de licence, vous conservez vos droits d'auteur et le droit d'utiliser vos propres contributions à toute autre fin.

Voici quelques exemples :

ALP pour les participants individuels :

- http ://apache.org/licenses/icla.txt
- http ://code.google.com/legal/individual-cla-v1.0.html

ALP pour les entreprises participantes :

- http ://apache.org/licenses/cla-corporate.txt
- http ://code.google.com/legal/corporate-cla-v1.0.html

Transfert du droit d'auteur

En transférant ses droits d'auteur, le participant accorde au projet la propriété de ses contributions. Il signe simplement un papier qu'il transmet, par fax ou voie postale, au projet.

Certains projets imposent une attribution complète des contributions. Si les termes de la licence Open Source doivent être défendus devant un tribunal, une personne morale possédant les droits sur tout le code sera en meilleure position pour défendre ses arguments. Sinon tous les participants pourraient être amenés à coopérer, mais certains risquent de ne pas avoir le temps, ou de ne pas être joignables, lorsque le problème se pose.

Selon les organismes, la rigueur appliquée à la collecte des droits varie. Certains se contentent d'une déclaration informelle du participant sur une liste de diffusion publique, quelque chose comme : « J'accorde, par la présente, les droits sur ce code au projet, les mêmes termes de licence que ceux du projet s'appliquant ». Selon au moins un avocat avec qui je me suis entretenu, cette déclaration est suffisante, certainement parce qu'elle est faite dans un cadre où l'attribution du droit d'auteur est normale et attendue de toute façon. De plus, c'est aussi un moyen pour le projet de s'assurer de la bonne foi du participant, de ses intentions profondes. La Free Software Foundation, de son côté, demande aux participants de signer personnellement, et d'envoyer par voie postale un document contenant une déclaration formelle d'attribution des droits d'auteur, parfois juste pour une contribution, parfois

pour toutes les contributions présentes et à venir. Si le développeur est employé par une entreprise, la FSF demande également que le document soit signé par l'employeur.

La paranoïa de la FSF est compréhensible. Si quelqu'un viole les termes de la GPL en incorporant du code sous licence GPL dans un programme propriétaire, il faut que la FSF puisse se défendre devant un tribunal, elle veut que ses droits d'auteur soient inattaquables. Puisque la FSF est détentrice des droits d'auteur de nombreux logiciels populaires, elle doit s'y préparer. En général, à moins que pour des raisons particulières votre projet nécessite une cession des droits en bonne et due forme, contentez vous des ALP, ils facilitent la vie de tout le monde.

6. La double liecence

Certains projets tentent de s'auto-financer en adoptant une double licence, c'est un système par lequel toute société peut payer le détenteur des droits d'auteur et obtenir son accord pour utiliser le code dans un programme propriétaire, le code restant libre pour les projets Open Source. Les librairies de code se prêtent évidemment mieux à cette double licence que des applications autonomes. Les termes exacts varient d'un cas à l'autre. La licence pour l'aspect libre est la GNU GPL, puisqu'elle empêche déjà n'importe qui d'ajouter le code protégé dans un produit propriétaire sans l'autorisation du détenteur des droits, parfois, c'est une licence personnalisée ayant les mêmes effets. Ainsi la licence MySQL [1] utilise la licence GNU GPL, on peut aussi citer comme exemple de licence personnalisée le système de licence de Sleepycat Software.

Vous vous demandez sûrement comment le détenteur des droits peut proposer une licence propriétaire contre rémunération si les termes de la GNU GPL stipulent que le code doit être mis à disposition sous des conditions moins restrictives. Les termes de la GPL sont imposés par le détenteur des droits à tout le monde sauf aux logiciels propriétaires, l'auteur est libre de décider de ne pas s'imposer les mêmes termes. Imaginez que le détenteur des droits possède un nombre infini de copies du logiciel rangées dans un seau, à chaque fois qu'il sort une copie du seau pour la donner au monde, il peut décider quelle licence appliquer : GPL, propriétaire ou autre. Ce n'est pas la GPL ou une quelconque autre licence Open Source qui lui donne ce droit, ce sont les lois du droit d'auteur.

Dans le meilleur des cas, la double licence permet ainsi au projet de logiciel libre de s'assurer une source de revenus stable. Malheureusement, la dynamique normale d'un projet Open Source peut s'en retrouver altérée. Le problème est que chaque volontaire apportant sa contribution au code participe maintenant aux deux entités : à la fois à la version libre et à la version propriétaire du code. Contribuer à la version libre ne lui posera certainement pas problème, c'est la norme dans les projets Open Source, mais par contre, participer à la création de revenus semi-propriétaires au bénéfice de tiers lui conviendra peut-être moins. C'est d'autant plus vrai que, pour une double licence, les participants doivent céder leurs droits au projet par une déclaration écrite. Le projet doit se couvrir au cas où un participant mécontent prétendrait à un pourcentage sur les revenus propriétaires. À partir du moment

1. http ://www.mysql.com/company/legal/licensing/

où les participants signent cette attribution des droits, ils ne peuvent plus ignorer que leur production enrichira une autre personne.

Tous les volontaires ne se soucieront pas de cela, après tout, leur contribution participe également au développement de l'édition Open Source, et c'est bien là leur intérêt. Néanmoins, la double licence est un exemple où le propriétaire des droits d'auteur se donne à lui-même un droit particulier que les autres personnes du projet n'ont pas, ce qui conduira tôt ou tard à des tensions, du moins avec quelques volontaires.

En pratique, il semble que les entreprises développant des logiciels à double licence n'ont pas une communauté de développement vraiment égalitaire. Ils profitent de correctifs de bogues mineurs ou de petits travaux de bénévoles, mais font le gros du travail en interne. Par exemple, Zack Urlocker, vice président du marketing chez MySQL, m'a confié que l'entreprise finit souvent pas employer les volontaires les plus actifs de toute façon. Ainsi, le produit en lui-même est Open Source, sous licence GPL. Son développement est plus ou moins contrôlé par une entreprise, même si la possibilité d'initier une fourche en cas de profond désaccord existe (elle reste extrêmement faible cependant). Je ne sais pas comment cette menace affecte la politique de l'entreprise, mais en tout cas, MySQL ne semble pas avoir de problème d'acceptation, que ce soit dans le monde de l'Open Source ou ailleurs.

7. Brevets

Les brevets logiciels sont LE sujet délicat du moment pour les logiciels libres car ils représentent la seule vraie menace contre laquelle la communauté ne peut se défendre. On peut toujours contourner les problèmes de droits d'auteur et de marque. Si une partie de votre code semble enfreindre le droit d'auteur de quelqu'un, il ne reste qu'à l'écrire de nouveau. S'il se trouve que quelqu'un a déjà les droits sur le nom de votre projet, dans le pire des cas vous pouvez toujours changer l'appellation. Même si cela est un désagrément temporaire, à long terme, c'est sans importance puisque le code en lui-même reste identique. Par contre, un brevet vous interdit purement et simplement d'implémenter une idée précise. La manière d'écrire le code ou même le langage utilisé importent peu. Dès qu'un projet de logiciel libre est accusé de violer un brevet, le projet doit : soit arrêter d'utiliser cette fonctionnalité particulière, soit faire face à un procès coûteux en temps et en argent. Puisqu'en général ces procès sont intentés par des firmes aux poches bien remplies, leur puissance financière leur permettant d'acheter les brevets, les projets de logiciel libre n'ont pas le luxe de cette dernière option et doivent immédiatement capituler, peu importe la validité du brevet face à un tribunal. Pour éviter tout simplement de se retrouver dans cette situation, les projets de logiciels libres se mettent à coder défensivement en fuyant les algorithmes brevetés, même s'ils représentent la meilleure, voire la seule, solution possible à un problème de programmation [1].

1. Sun Microsystems et IBM se sont montrés de bons alliés dans cette bataille en libérant un grand nombre de brevets logiciels, 1600 et 500 respectivement, pour que la communauté Open Source puisse en profiter. N'étant pas avocat, je ne saurais évaluer la portée de ces dons. Mais, quand bien même ces brevets seraient importants et que les termes de cette libération les rendraient vraiment libres d'utilisation par n'importe quel projet Open Source, il n'en reste pas moins que cela ne représente qu'une goutte d'eau dans l'océan.

Des sondages et l'expérience montrent que non seulement une vaste majorité des programmeurs Open Source, mais aussi de tous les programmeurs, pense que les brevets logiciels devraient être abolis, purement et simplement [1]. Les programmeurs Open Source les voient d'un très mauvais œil, et parfois refusent de travailler sur des projets qui naviguent dans les eaux troubles de la collecte ou de la mise en application des brevets logiciels. Si votre organisation amasse des brevets logiciels, annoncez publiquement, clairement et de manière irrévocable que ces brevets ne seront jamais employés contre un projet Open Source, et qu'ils ne seront utilisés qu'à des fins défensives au cas où une tierce partie lancerait une action en justice pour violation de brevet contre votre organisation. Ce n'est pas seulement votre unique alternative, c'est aussi un exemple de bonne communication dans le monde de l'Open Source [2].

Malheureusement, la collecte de brevets à des fins défensives est une action rationnelle. Le système de brevets actuel, aux États-Unis en tout cas, est de par sa nature une course à l'armement : si vos adversaires ont acquis de nombreux brevets, alors votre meilleure défense est d'en acquérir énormement à votre tour, ainsi, si l'on vous attaque pour violation de brevet, vous pouvez répondre par la même menace. En conséquence, les deux parties concernées se réuniront à la table des négociations et s'entendront sur un accord de licence croisée. Dès lors, personne n'aura rien à payer, sauf aux avocats spécialisés en propriété industrielle bien sûr.

Les dommages causés aux logiciels libres par les brevets logiciels sont plus insidieux que la simple menace sur le développement. Les brevets logiciels encouragent au secret chez les développeurs de firmware qui craignent, à juste titre, qu'une publication des détails de leur interface ne fournisse une aide technique aux adversaires ne cherchant qu'à les attaquer pour violation de brevet logiciel. Tout ceci est loin de n'être que théorique, le secteur des cartes graphiques en est un bon exemple. Beaucoup de fabricants de cartes graphiques sont réticents à l'idée de publier les spécifications détaillées de leurs programmes. Elles sont pourtant nécessaires à l'écriture de pilotes Open Source de très bonne qualité pour leur matériel. Le support de ces cartes par les systèmes d'exploitation libres est donc largement incomplet. Pourquoi les fabricants font-ils cela ? Pourquoi empêcheraient-ils le support de leurs produits ? Après tout, si leurs cartes devenaient compatibles avec davantage de systèmes d' exploitation, cela se traduirait par de meilleures ventes. C'est surtout un problème de discrétion. Dans le secret des salles de fabrication, ces vendeurs ne cessent de violer la propriété intellectuelle de leurs concurrents, parfois sciemment, parfois par accident. Les brevets sont si imprévisibles, et couvrent parfois des domaines si vastes, qu'aucun fabricant de carte ne peut jamais être sûr de ne violer aucun brevet logiciel, même après une recherche d'antériorité. Par conséquent, les producteurs n'osent pas publier les spécifications complètes de leurs interfaces de peur de fournir à leurs adversaires le bâton pour se faire battre. (Évidemment, le sujet étant sensible, vous ne trouverez aucun témoignage écrit d'une source sûre le confirmant, je l'ai appris grâce à un contact personnel.)

1. Voir l'enquête « Software Developers on Patent Law » (http ://-groups.csail.mit.edu/mac/projects/lpf/Whatsnew/survey.html) pour un exemple d'un de ces sondages.

2. Par exemple, RedHat a juré de ne jamais utiliser ses brevets contre un projet Open Source, voir http ://www.redhat.com/legal/patent_policy.html.

Certaines licences de logiciel libre possèdent des clauses particulières pour combattre, ou au moins décourager, les brevets logiciels. La GNU GPL, par exemple, dit ceci :

> 7. Si, en conséquence d'un jugement du tribunal ou d'allégations de violation de brevet ou pour toute autre raison (pas nécessairement liée aux brevets), des restrictions vous sont imposées (que ce soit par ordre du tribunal, accord ou pour toute autre raison) et que ces restrictions ne sont pas compatibles avec les conditions de cette Licence, elles ne vous dégagent pas des obligations de cette Licence. Si vous ne pouvez pas distribuer le Logiciel en conciliant le respect de cette Licence et toutes autres obligations pertinentes, alors vous ne pourrez plus distribuer le Logiciel. Par exemple, si une licence de brevet ne vous autorisait pas une redistribution libre de droits du Programme par tous ceux qui en reçoivent une copie directement ou indirectement de vous, alors le seul moyen de satisfaire à cette obligation et aux obligations de la Licence serait de cesser la distribution du programme complètement. [...] Cette section n'est en rien une incitation à la violation de brevets ou de tout autre droit de propriété, pas plus qu'il ne vous encourage à contester la validité des droits qui vous sont opposés. Le seul et unique objet de cette partie est de protéger l'intégrité des systèmes de distribution de logiciels libres qui reposent sur les licences publiques. Ces licences sont un gage de confiance, de pérennité. Grâce à elles, de nombreuses personnes croyant en ce système de distribution ont pu faire de généreuses donations à d'innombrables projets. Il revient à l'auteur ou au donateur de décider s'il désire distribuer le logiciel par un autre système et une licence ne peut pas imposer ce choix.

La Licence Apache [1], dans sa version 2.0, contient également des pré-requis contre les brevets. Premièrement, elle stipule que toute personne distribuant le code sous cette licence doit y inclure implicitement une licence de brevet sans redevance pour tout brevet qu'elle pourrait détenir et qui pourrait s'appliquer au code. Deuxièmement, ce qui est plus ingénieux encore, elle punit quiconque initierait un procès pour violation de brevet sur le travail couvert en rompant automatiquement la licence de brevet implicite dès le moment où une telle revendication est faite :

> 3. Attribution d'une licence de brevet. Soumis aux termes et conditions de cette licence, chaque Participant, par la présente, Vous accorde un droit d'exploitation de brevet impérissable, universel, non-exclusif, sans frais, sans redevance, irrévocable (à l'exception des conditions décrites dans cette section) pour l'usage présent et passé,

1. http ://www.apache.org/licenses/LICENSE-2.0

> l'utilisation, la proposition à la vente, la vente, l'import et pour tout transfert de l'Oeuvre. Cette licence ne vaut que pour les revendications de brevet que le participant peut licencier et qui sont nécessairement violées par leur(s) Contribution(s) seule(s) ou par une combinaison de leur(s) Contribution(s) et de l'Oeuvre à laquelle une (des) telle(s) Contribution(s) a (ont) été soumise(s). Si vous établissez un contentieux contre une tierce partie (demande entre défenseurs ou demande reconventionnelle dans un procès comprises) alléguant que l'Oeuvre ou qu'une Contribution incluse dans l'Oeuvre constitue une violation directe ou contributive de toute licence de brevet, alors toutes les brevets Vous ayant été attribués par les termes de cette Licence pour cette Oeuvre deviendraient nuls à la date où ce contentieux est enregistré.

Aussi importante soit-elle, aussi bien légalement que politiquement, la défense érigée au sein des licences de logiciels libres contre les brevets ne suffira pas à repousser la menace que font peser les actions légales sur les logiciels libres. Seule une modification des lois internationales sur les brevets, dans les textes ou dans leur interprétation, pourrait assurer la sécurité des logiciels libres. Pour en savoir plus sur les brevets logiciels et les combats menés, rendez-vous sur nosoftwarepatents.com [1]. L'article de Wikipédia [2] contient également beaucoup d'informations utiles sur les brevets logiciels.

8. Plus de références

Ce chapitre ne vous dévoile qu'une partie des enjeux des licences pour logiciels libres. Même si, comme je l'espère, il est assez complet pour vous permettre de commencer votre propre projet Open Source, une recherche sérieuse sur ce thème vous fournira sans problème plus d'informations que ne peut le faire ce livre. Voici une liste d'autres références sur les licences Open Source :

- *Understanding Open Source and Free Software Licensing* par Andrew St. Laurent. Publié par O'Reilly Media, première édition Août 2004, ISBN : 0-596-00581-4. C'est un livre complet qui traite des licences Open Source dans toute leur complexité, y compris pour les questions éludées dans ce chapitre.
- *Make Your Open Source Software GPL-Compatible. Or Else.* par David A. Wheeler [3]. Il s'agit d'un article détaillé et bien rédigé sur l'importance du choix d'une licence compatible avec la GPL même si vous n'utilisez pas cette dernière. L'article aborde également bien d'autres questions relatives aux licences et contient de nombreux liens intéressants.

1. http ://www.nosoftwarepatents.com/
2. http ://fr.wikipedia.org/wiki/Brevet_logiciel
3. http ://www.dwheeler.com/essays/gpl-compatible.html

– Creative Commons [1]. Creative Commons est une organisation qui promeut toute une gamme de droits d'auteurs plus libres et plus flexibles que ce qu'encourage la pratique traditionnelle du droit d'auteur. Elle ne propose pas uniquement des licences pour logiciels, mais aussi pour l'écrit, l'art et la musique, tous accessibles par un sélectionneur de licence facile d'emploi. Certaines de ces licences sont des copylefts, d'autres non, malgré leur gratuité, d'autres encore traitent du droit d'auteur conventionnel auquel certaines restrictions ont été ôtées. Le site Web Creative Commons [2] propose des explications parfaitement claires sur chacune des licences. Si vous deviez choisir un site pour démontrer les implications philosophiques étendues du mouvement des logiciels libres, c'est celui que je vous conseille.

1. http ://creativecommons.org/
2. http ://creativecommons.org/

ANNEXE A

Solutions libres de gestion de versions

Voici une liste de tous les logiciels Open Source de gestion de versions à ma connaissance à la mi-2007. Le seul que j'utilise régulièrement est Subversion. Subversion et CVS mis à part, je vous confesse que mon expérience de ces systèmes est limitée, voire inexistante. Les informations présentées ici sont tirées des sites Web des logiciels. Voir également les pages Wikipedia consacrées en anglais [1] ou en français [2].

CVS — Voici quelque temps maintenant que CVS [3] existe, de nombreux développeurs le connaissent déjà bien. À son époque, il était révolutionnaire : il fut le premier logiciel de gestion de versions Open Source à proposer aux développeurs un large accès réseau (à ma connaissance), et le premier à permettre aux utilisateurs anonymes d'extraire le code, facilitant l'implication en douceur de nouveaux développeurs dans un projet. CVS ne versionne que les fichiers, pas les dossiers. Il permet de créer des branches, des étiquettes et propose de bonnes performances côté client, mais il ne gère pas très bien les gros fichiers ou les fichiers binaires. Les *commits* atomiques ne sont pas non plus pris en charge. [Note : j'ai participé au développement de CVS pendant cinq ans environ, puis j'ai rejoint le projet Subversion créé pour le remplacer.]

1. http ://en.wikipedia.org/wiki/List_of_revision_control_software
2. http ://fr.wikipedia.org/wiki/Logiciel_de_gestion_de_versions
3. http ://nongnu.org/cvs/

Subversion — Subversion [1] a été créé pour remplacer CVS. Il présente sensiblement la même approche du contrôle de versions, mais corrige les problèmes que rencontraient fréquemment les utilisateurs de CVS et ajoute des fonctionnalités. L'un des buts de Subversion est de ne pas brusquer les habitués de CVS. Je ne m'attarde pas ici sur les fonctionnalités de Subversion, consultez le site Web du projet pour plus d'informations. [Note : Je suis impliqué dans le développement de Subversion, et c'est le seul système parmi ceux listés ici que j'utilise régulièrement.]

SVK — Bien que basé sur Subversion, SVK [2] ressemble probablement davantage à Arch. SVK gère le développement distribué (non-centralisé), possède un système de fusion des modifications intelligent, et a la capacité de copier des dépôts provenant de systèmes de gestion de versions autres que SVK. Référez-vous au site Web pour en savoir plus.

Mercurial — Mercurial [3] est un système de contrôle de versions distribué qui offre, entre autres, « un indexage croisé complet des fichiers et des ensembles de modifications, des protocoles de synchronisation HTTP et SSH optimisant l'utilisation de bande passante et de CPU, un système de fusion arbitraire entre branches, une interface Web indépendante. Il est de plus portable sur Unix, MacOS X et Windows » (cette liste de fonctionnalités est tirée du site Web de Mercurial).

GIT — GIT [4] est un projet initié par Linus Torvalds pour gérer l'arborescence du noyau Linux. Il cible très précisément les besoins du développement d'un noyau. Il était toujours en développement au moment de l'écriture de ce livre. Il ne semble pas encore disposer d'un site Web mais consultez l'entrée Wikipedia qui le concerne, les informations qu'elle contient devraient être à jour.

Bazaar — Bazaar [5] est encore en développement. Il sera une implémentation du protocole GNU Arch, en assurera la compatibilité à mesure qu'il évoluera et collaborera avec le groupe de travail de la communauté GNU Arch pour toute modification du protocole qui pourrait être nécessaire aux utilisateurs.

Darcs — « David's Advances Revision Control System [6] est un *n*ième remplaçant de CVS. Il est écrit en Haskell et portable sous Linux, MacOs X, FreeBSD, OpenBSD et Microsoft Windows. Darcs inclut un script cgi qui peut être utilisé pour visionner le contenu de votre dépôt. »

Arch — GNU Arch [7] supporte à la fois le développement distribué et centralisé. Les développeurs *commitent* leurs modifications dans une « archive », qui peut être locale, et les modifications peuvent être poussées ou insérées dans d'autres archives, selon les décisions du responsable de ces archives. Il en découle qu'Arch dispose d'un support de fusions plus sophistiqué que CVS. Arch permet également à une personne de créer facilement des branches pour des archives sur lesquelles elle n'a pas les droits

1. http ://subversion.tigris.org/
2. http ://svk.elixus.org/
3. http ://www.selenic.com/mercurial/
4. http ://en.wikipedia.org/wiki/Git
5. http ://bazaar.canonical.com/
6. http ://abridgegame.org/darcs/
7. http ://www.gnu.org/software/gnu-arch/

de *commit*. Ceci n'est qu'un simple aperçu des fonctionnalités d'Arch, voir les pages Web du projet pour plus d'informations.

Monotone — « Monotone est un système de gestion de versions libre. Il propose un système de « stockage de version » simple, utilisant un fichier unique et des transactions. Les opérations s'effectuent hors ligne et le système utilise un protocole de synchronisation pair-à-pair efficace. Il supporte la fusion de versions basée sur l'historique, un système de branches simplifié, la vérification de code intégrée et les tests externes. Il utilise des algorithmes de hashage pour l'identification des versions et des certificats RSA côté client. L'internationalisation est poussée et le programme ne possède pas de dépendances externes (fonctionne sous Linux, Solaris, OSX et Windows). Il est placé sous licence GNU GPL. »

Codeville — « Pourquoi un système de contrôle de versions de plus ? Tous les autres dispositifs vous demandent de suivre consciencieusement les différentes branches pour ne pas avoir à fusionner plusieurs fois les mêmes conflits. Codeville [1] est beaucoup plus anarchique. Il vous permet de mettre à jour ou de valider des modifications depuis n'importe quel dépôt, n'importe quand, sans fusions inutiles [...] Codeville crée un identifiant pour tout changement apporté, et garde en mémoire la liste de tous les changements appliqués à chaque ficher ainsi que le dernier changement pour chaque fichier et pour chaque ligne. Quand il y a un conflit, il vérifie si l'un des deux côtés a déjà été appliqué à l'autre et, si c'est le cas, déclare vainqueur l'autre côté automatiquement. Quand il y a effectivement un conflit de versions empêchant la fusion automatique, Codeville se comporte presque comme CVS. »

Vesta — « Vesta [2] est un système de gestion de versions portable qui vise à accompagner le développement de logiciels, quelle que soit leur taille, des plus petits (moins de 10 000 lignes de code) aux très gros (10 000 000 de lignes de code). [...] Vesta est un système mature, résultant de dix années de recherche et développement au centre de recherche Compaq/Digital System et a été employé par le groupe Compaq's Alpha Microprocessor pendant deux ans et demi. Le groupe Alpha était alors composé de plus de 150 développeurs répartis sur deux sites à des milliers de kilomètres de distance, sur les côtes Est et Ouest des États-Unis. Ont ainsi été gérées des distributions contenant jusqu'à 130 MO de données sources, chacune produisant 1,5 GO de données dérivées. Les versions compilées sur le site de la côte Est produisaient environ 10-15 GO de données dérivées par jour, toutes gérées par Vesta. Bien que Vesta ait été pensé autour du développement de logiciel, le groupe Alpha a démontré la flexibilité du système en l'utilisant pour développer du matériel informatique, alimentant l'outil avec les fichiers de données matériel en langage spécifique puis en construisant des simulateurs et d'autres objets dérivés avec l'outil de développement adossé à Vesta. Les membres de l'ancien groupe Alpha (incorporé à Intel maintenant) continuent d'utiliser cet outil dans le cadre d'un nouveau projet de microprocesseur. »

Aegis — « Aegis [3] est un système de gestion de versions. Il fournit un cadre de travail dans lequel une équipe de développement peut progresser indépendamment sur de nombreuses modifications d'un programme, Aegis prenant en charge l'intégration

1. http ://codeville.org/
2. http ://vestasys.org/
3. http ://aegis.sourceforge.net/

de ces modifications dans les sources de références avec aussi peu de perturbations que possible. »

CVSNT — « CVSNT [1] est un système de contrôle de versions multi-plateforme avancé. Compatible avec CVS, la référence en entreprise, il le complète de nombreuses fonctionnalités. ... CVSNT est Open Source (logiciel libre en licence GNU General Public License) ». Ses fonctionnalités incluent l'authentification via tous les protocoles standards de CVS auxquelles s'ajoutent les SSPI spécifiques Windows et Active Directory ; les échanges sécurisés par sserver ou SSPI chiffrés ; il est multi-plateforme (supporté en environnement Windows et Unix) ; l'intégration de la version NT aux systèmes Win32 est complète ; le système MergePoint supprime le besoin de baliser avant de fusionner ; bénéficie d'un développement actif.

META-CVS — « Meta-CVS [2] est un système de contrôle de versions construit autour de CVS. Bien qu'il conserve la plupart des fonctionnalités de CVS, y compris son protocole réseau, il est plus performant et plus simple ». Les fonctionnalités listées sur le site Web de META-CVS comprennent : le versionnement de la structure des dossiers, une gestion améliorée des formats de fichiers, un système de branche et de fusion simplifié et agréable, le support des liens symboliques, des listes de propriétés attachées aux données versionnées, un système d'importation de données extérieures et de migration depuis un référentiel CVS.

Open CM — « OpenCM [3] a été conçu pour être une alternative sécurisée à CVS en assurant un haut niveau d'intégrité des transactions. La page « Fonctionnalités » liste un certain nombre de fonctions essentielles. Bien qu'il ne soit pas aussi complet que CVS, il supporte quelques trucs utiles qui manquent à CVS. Pour faire court, OpenCM fournit un support de qualité pour renommer et configurer, une authentification et un contrôle d'accès chiffrés ainsi qu'un support avancé des branches. »

PRCS — « PRCS [4], le *Project Revision Control System*, est l'interface d'une collection d'outils qui, comme CVS, permettent de gérer des ensembles de fichiers et de dossiers comme une entité propre, en leur assurant une cohérence de version. [...] Son but est proche de celui de SCCS, RCS et CVS, mais (d'après leurs auteurs en tout cas), il est beaucoup plus accessible que n'importe lequel de ces systèmes.

ArX — ArX [5] est un système de contrôle proposant des fonctionnalités pour gérer les branches et les fusions, ou pour assurer l'intégrité des données par chiffrement. Il permet de publier facilement des archives sur un simple serveur HTTP.

Source — « SourceJammer [6] est un système de contrôle de sources et de versions écrit en Java. Il consiste en un composant serveur qui gère les fichiers, l'historique de versions, traite les réservations, validations et autres commandes, et possède un composant client qui exécute les requêtes du serveur et gère les fichiers sur le système de fichier du client. »

1. http ://cvsnt.org/
2. http ://users.footprints.net/%7Ekaz/mcvs.html
3. http ://www.opencm.org/
4. http ://prcs.sourceforge.net/
5. http ://www.nongnu.org/arx/
6. http ://sourcejammer.org/

FastCST — « FastCST [1] est un système 'moderne' qui manipule des révisions de fichiers groupées, et utilise des opérations distribuées plutôt qu'un contrôle centralisé. Il suffit d'une adresse e-mail pour utiliser FastCST. Pour les configurations plus importantes, vous n'avez besoin que d'un simple serveur FTP et/ou d'un serveur HTTP, ou d'utiliser la commande intégrée 'serve' pour exposer directement votre référentiel. Tous les ensembles de modifications sont uniques et renferment des tonnes de meta-données pour pouvoir rejeter tout ce dont vous ne voulez pas avant d'importer les données. La fusion s'opère par comparaison d'un ensemble de modifications à fusionner avec le contenu actuel du référentiel plutôt qu'avec un autre ensemble de modifications. »

Supervision — « Supervision [2] est un système de contrôle de versions multi-utilisateur basé sur les ensembles de modifications. Son but est de proposer une alternative Open Source de niveau professionnel aux solutions commerciales, aussi simple à utiliser (voire même plus simple) et tout aussi puissante. De fait, une prise en main intuitive et efficace a été l'une des grandes priorités dans le développement de Supervision, et ce dès le début du projet. »

1. http ://www.zedshaw.com/projects/fastcst/index.html
2. http ://www.supervision.org/

ANNEXE B

Systèmes libres de suivi de bogues

Quel que soit le système de suivi de bogues qu'un projet choisit, il y a aura toujours quelqu'un pour s'en plaindre. Ce phénomène n'est pas nouveau, et aucun outil de développement partagé n'y échappe, mais il se fait encore plus sentir dans ce cas particulier. Je pense que c'est dû au fait qu'un tel outil est très visuel et interactif. Il paraît facile d'imaginer les améliorations qu'on pourrait y apporter (si seulement on en avait le temps) et de les décrire à tout le monde. Il vous faudra donc accepter les inévitables griefs avec philosophie. La plupart des systèmes proposés ci-dessous sont plutôt bons.

Tout au long de cette liste, le mot « problème » sera utilisé pour décrire les objets répertoriés dans le système de suivi. Mais souvenez vous que chaque système peut employer sa propre terminologie, ainsi vous pourrez également retrouver « artéfact » ou « bogue » ou d'autres termes encore.

Bugzilla — Bugzilla [1] est très populaire, maintenu activement et semble apprécié des utilisateurs. La version modifiée que j'emploie depuis quatre ans me satisfait tout à fait. Il n'est pas très personnalisable, mais c'est peut-être étrangement l'un de ses avantages : les installations de Bugzilla sont à peu près similaires partout : les développeurs sont déjà habitués à son interface et restent donc en terrain connu.

1. http ://www.bugzilla.org/

GNATS — GNU GNATS [1] est un système de suivi de bogues des plus anciens et des plus largement répandus. Ses points forts sont la diversité des interactions avec le programme (il peut être utilisé non seulement grâce à un navigateur Web, mais aussi par e-mail ou en ligne de commande), ainsi que le stockage des problèmes en texte brut. Le fait que tous les problèmes soient stockés sous forme de fichiers textes sur disque rend plus aisée l'écriture d'outils d'analyse des données (par exemple, afin de générer des rapports statistiques). GNATS sait aussi traiter les e-mails par différents moyens et les mettre en relation avec les problèmes correspondants en se basant sur l'en-tête. L'enregistrement des conversations utilisateurs/développeurs est, de ce fait, très simple.

RequestTracker (RT) — RT [2] est présenté comme « un système professionnel permettant à un groupe de personnes de gérer intelligemment et efficacement les tâches, les problèmes et les demandes émanant de la communauté des utilisateurs », ce qui est un résumé plutôt bon. RT dispose d'une interface Web assez léchée, et semble bénéficier d'une large base d'utilisateurs. L'interface est un peu complexe visuellement, mais on s'y habitue et elle gêne moins avec le temps. RT est sous licence GNU GPL (étrangement, ce n'est pas clairement affiché sur le site Web).

Trac — Trac [3] est un peu plus qu'un système de suivi de bogues : c'est plutôt un wiki couplé à un système de référencement de bogues. Il utilise les liens wiki pour connecter les problèmes, les fichiers, les modifications sous contrôle de versions et les pages wiki classiques. Il est simple à mettre en œuvre et s'intègre à Subversion (voir l'Annexe A).

Roundup — Roundup [4] est assez facile à installer (le seul pré-requis est Python 2.1 ou supérieur) et simple d'utilisation. Il propose des interfaces Web, e-mail et par ligne de commande. Les formulaires de données des problèmes et l'interface Web sont personnalisables, tout comme une partie de sa logique de changement d'état.

Mantis — Mantis [5] est une application Web de système de suivi de bogues écrite en PHP, utilisant MySQL pour le stockage des données. Elle possède les fonctionnalités que vous attendez d'un système de référencement de bogues. Personnellement, je trouve l'interface propre, intuitive et agréable à l'œil.

Flyspray — Flyspray [6] est une application Web de suivi de bogues écrite en PHP. Sur sa page Web, elle est décrite comme « décompliquée » et parmi ses fonctionnalités, on retrouve : le support de bases de données multiples (MySQL et PGSQL actuellement), la gestion de projets multiples, la surveillance de tâches avec des alertes de modifications (par e-mail ou Jabber), un historique complet des tâches, des thèmes par feuilles de style CSS, l'ajout de pièces jointes, des outils de recherche avancés (mais toujours simple à utiliser), le support des flux RSS/Atom, la saisie au format wiki ou texte brut, le vote et la représentation graphique des dépendances.

1. http ://www.gnu.org/software/gnats/
2. http ://www.bestpractical.com/rt/
3. http ://trac.edgewall.com/
4. http ://roundup.sourceforge.net/
5. http ://www.mantisbt.org/
6. http ://www.flyspray.org/

Scarab — L'idée de base de Scarab [1] est d'offrir un système de suivi de bogues très complet et personnalisable. Il propose plus ou moins de rassembler les fonctionnalités assurées par les autres systèmes de référencement de bogues : l'entrée de données, les requêtes, les rapports, les notifications aux personnes concernées, la gestion collaborative des commentaires et le suivi des dépendances. La personnalisation se fait au travers de pages Web d'administration. Vous pouvez multiplier les « modules » (projets) actifs au sein d'une même installation de Scarab. Dans un module donné, vous pouvez créer de nouveaux types de problèmes (défauts, améliorations, tâches, requêtes de support, etc.) et ajouter des attributs arbitraires pour adapter le système de référencement aux besoins spécifiques de votre projet. Vers fin 2004, Scarab se rapprochait de la sortie de sa version 1.0.

Debian Bug Tracking System (DBTS) — Le Debian Bug Tracking System est surprenant par le fait que toutes les saisies ou manipulations de problèmes se font par e-mail : chaque problème se voit attribuer sa propre adresse email. Le DBTS s'adapte très bien à la taille de n'importe quel projet ; bugs.debian.org [2] recense par exemple 277 741 problèmes. Comme toutes les interactions se font par le client de messagerie classique, un environnement familier et facilement accessible pour la plupart des personnes, le DBTS est particulièrement adapté à l'enregistrement de nombreux rapports qui nécessitent tri et prompte réaction. Il n'est évidemment pas non plus exempt de défauts. Les développeurs doivent passer par une étape d'apprentissage du système de commandes par e-mail, et les utilisateurs doivent écrire leurs rapports de bogue sans pouvoir s'inspirer d'un formulaire en ligne pour connaître les informations à transmettre. Il existe des outils pour aider les utilisateurs à envoyer de meilleurs rapports de bogues tels que le programme de rapport de bogue en ligne de commande ou le paquet debbugs-el pour Emacs. Mais la plupart des gens n'utiliseront pas ces outils, ils se contenteront d'écrire les e-mails à la main et suivront (ou pas) les consignes pour écrire un bon rapport de bogue données par votre projet. Le DBTS propose une interface Web en lecture seule pour la consultation et la recherche de problèmes.

Suivi des dossiers d'incident

Ces programmes sont plus axés sur le suivi de l'assistance que sur le suivi des bogues. Un système de référencement de bogues sera sans aucun doute plus adapté. Je vais aussi lister ces programmes afin d'être exhaustif et parce qu'un projet exotique tire meilleur parti d'un système de suivi des dossiers d'incidents que d'un système de référencement de bogues classique.

WebCall — Voir le descriptif de Webcall [3] sur le site officiel.

Bluetail Ticket Tracker (BTT) — BTT [4] se situe à mi-chemin entre le suivi des dossiers d'incidents et le système de référencement de bogues. Il propose des options de

1. http ://scarab.tigris.org/
2. http ://bugs.debian.org/
3. http ://myrapid.com/webcall/
4. http ://btt.sourceforge.net/

confidentialité qui sont plutôt rares parmi les systèmes de référencement de bogues libres : les utilisateurs sont rangés dans les catégories Personnel, Ami, Client ou Anonyme, et la quantité d'information disponible dépend de la catégorie à laquelle vous appartenez. On retrouve l'intégration des e-mails, une interface en ligne de commande et des mécanismes de conversion des e-mails en dossiers. Il propose également des outils pour suivre des informations qui ne sont pas liées à des dossiers particuliers comme la documentation interne ou les FAQs.

ANNEXE C

Pourquoi se soucier de la couleur de l'abri à vélos

Vous ne devriez pas : ce n'est pas vraiment important et vous avez mieux à faire. Le fameux message « L'abri à vélos » de Poul-Henning Kamp (le chapitre 6, en contient un extrait) est une métaphore éloquente de ce qui peut mal tourner dans une discussion de groupe. Il est reproduit ici avec son autorisation [1].

Sujet : Un abri à vélos (peu importe la couleur), sur de l'herbe plus verte...

De : Poul-Henning Kamp <phk@freebsd.org>

Date : Samedi, 2 Oct 1999 16 :14 :10 +0200

Message-ID : <18238.938873650@critter.freebsd.dk>

Expéditeur : phk@critter.freebsd.dk

Bcc : Liste de distribution invisible ;

MIME-Version : 1.0

[Copie envoyée aux committers et hackers]

1. L'adresse originale est : http ://www.freebsd.org/cgi/getmsg.cgi ?-fetch=506636+517178+/usr/local/www/db/text/1999/freebsd-hackers/ 19991003.freebsd-hackers.

Mon dernier pamphlet a été suffisamment bien reçu pour ne pas me décourager d'en écrire un autre, or aujourd'hui, j'ai le temps et l'envie de le faire. J'ai eu un peu de mal à me décider concernant la liste des destinataires, mais cette fois j'ai mis en copie tous les committers et les hackers et je pense que c'est le meilleur choix. Je ne suis pas inscrit sur la liste des hackers, mais on en reparlera. Ce qui m'a décidé cette fois, est le sujet « sleep(1) devrait faire des secondes fractionnées » qui nous pourrit la vie depuis plusieurs jours maintenant, voire même plusieurs semaines, je n'ai même pas envie de prendre la peine de vérifier.

Si vous avez manqué ce sujet : Félicitations.

La proposition qui a mis le feu aux poudres est celle de rendre sleep(1) DTRT si l'argument entré n'est pas un nombre entier. Je ne m'étendrai pas sur le sujet, ce n'est vraiment pas aussi important que ne le laissent penser les nombreuses réponses, et il a déjà reçu bien plus d'attention que certains des « vrais problèmes » qu'on a ici.

La saga sleep(1) est l'exemple le plus marquant d'une discussion sur la couleur de l'abri à vélos que nous ayons eu sur FreeBSD. La proposition en elle-même est bonne, cela nous permettrait d'être compatible avec OpenBSD et NetBSD tout en gardant la compatibilité avec chaque morceau de code déjà écrit. Et pourtant, il y eut tellement d'objections, d'idées et de modifications soulevées et proposées, qu'on eut cru que le changement allait boucher les trous d'un fromage suisse, modifier le goût du Coca Cola ou quelque chose d'aussi sérieux.

« Quel rapport avec l'abri à vélos ? » demandez-vous ?

C'est une longue histoire, ou plutôt une vieille histoire, mais très brève en réalité. C. Northcote Parkinson écrivit un livre dans les années 60 intitulé « La Loi de Parkinson » qui contient des opinions intéressantes sur la dynamique du management. Vous pouvez le trouver sur Amazon et peut-être dans la bibliothèque de votre père. En tout cas, il vaut bien son prix et vous ne regretterez pas le temps passé à le lire, et si vous aimez Dilbert vous aimerez Parkinson. Quelqu'un m'a récemment dit l'avoir lu et trouver que seuls 50% environ de ses préceptes peuvent encore s'appliquer aujourd'hui. Je trouve cela carrément bien, beaucoup de livres modernes sur le management ont un taux de réussite bien plus faible, or ce livre à plus de 35 ans. Dans cet exemple précis impliquant l'abri à vélos, l'autre élément important est une centrale nucléaire, je pense que cela illustre bien l'âge de l'ouvrage.

Parkinson montre comment vous pouvez obtenir du bureau de direction l'accord de construction d'une centrale nucléaire coûtant plusieurs millions, voire un milliard de dollars, mais que, si vous voulez construire un abri à vélos, vous vous trouvez empêtré dans des discussions sans fin. Parkinson explique qu'une centrale nucléaire est si vaste, si chère et si complexe que les gens ne peuvent l'appréhender et, plutôt que d'essayer, ils préfèrent supposer qu'une autre personne a pris la peine de vérifier tous les détails avant d'aller plus loin. Richard P. Feynmann donne, dans son livre, quelques exemples intéressants et très pertinents concernant Los Alamos. À côté, nous avons l'abri à vélos. N'importe qui peut en construire un en un week-end, sans pour autant manquer le match à la télé. Aussi bien préparée et raisonnable que soit votre proposition, quelqu'un saisira cette opportunité pour montrer qu'il fait son boulot, qu'il veille au grain, qu'il est bien présent. Au Danemark, on appelle cela : laisser son empreinte. C'est une question d'amour propre et de prestige, la fierté de pouvoir désigner quelque chose du doigt en disant : « Voilà ! C'est moi qui l'ai fait. » C'est un trait de caractère très présent chez les politiciens, mais également chez toute personne à qui on donne un tout petit peu de pouvoir. Il n'y a qu'à voir les traces de pas dans le ciment frais.

J'ai beaucoup de respect pour celui qui a proposé l'idée à l'origine. Il a défendu sa vision becs et ongles, et la modification est intégrée dans notre arbre maintenant. J'aurai abandonné après seulement quelques messages dans ce sujet.

Ce qui nous amène à, comme promis précédemment, la raison pour laquelle j'ai quitté la liste -hackers : je me suis désinscrit de -hackers, il y a quelques années, parce que je ne pouvais plus suivre la quantité d'e-mails reçus. Depuis j'ai quitté d'autres listes pour la même raison. Et je reçois toujours beaucoup d'e-mails. Une grande partie est directement redirigée vers /dev/null par des filtres : des gens comme [masqué] n'apparaîtront jamais sur mon écran, les ajouts aux documents dans les langages que je ne comprends pas non plus, les ajouts à des portages : idem. Toutes ces choses passent sans que je ne les vois jamais. Mais malgré ces pièges acérés protégeant ma boîte aux lettres, je reçois toujours trop d'e-mails.

J'en arrive donc à l'herbe plus verte :

J'aimerais que l'on puisse réduire le bruit de fond sur nos listes, et j'aimerais qu'on laisse les gens construire des abris à vélos de temps en temps, et je me moque pas mal de la couleur utilisée pour les peindre.

Le premier de mes souhaits relève de la civilité, de la compréhension et de l'intelligence dans notre usage des e-mails. Si je pouvais définir un ensemble de critères, de manière précise et concise, pour que les gens sachent quand répondre ou ne pas répondre à un e-mail, que tout le monde approuve et s'y tienne, je serai un homme heureux. La sagesse me dit cependant que je perdrais mon temps. Mais je peux vous suggérer quelques fenêtres que j'aimerais voir apparaître dans les outils de messagerie, et qui s'ouvriraient quand les gens envoient ou répondent à un e-mail sur une liste à laquelle je devrais m'inscrire :

```
+---------------------------------------+
| Votre e-mail va être envoyé à plusieurs
| centaines de milliers de personnes qui
| devront prendre au moins 10 secondes
| pour décider de son intérêt. Cela
| représente l'équivalent de deux semaines
| de lecture. La plupart des destinataires
| devront en plus payer pour télécharger
| votre e-mail.
| Êtes-vous vraiment sûr que votre e-mail
| est d'une importance suffisante pour
| déranger tous ces gens?
|      [OUI] [MODIFIER] [ANNULER]
+---------------------------------------+

+---------------------------------------+
| Attention : Vous n'avez peut-être pas
| encore lu tous les emails de ce sujet.
| Quelqu'un d'autre peut déjà avoir dit
| ce que vous vous apprêtez à dire dans
| votre réponse. Veuillez lire le sujet
```

```
| entièrement avant de répondre.
|             [ANNULER]
+--------------------------------------+

+--------------------------------------+
| Attention : Votre programme de
| messagerie ne vous a même pas
| encore montré tout le message. Par
| conséquent vous n'avez pas pu le lire
| et le comprendre entièrement.
| Il est impoli de répondre à un e-mail
| avant de l'avoir entièrement lu et d'y
| répondre dans l'heure qui suit.
|             [ANNULER]
+--------------------------------------+

+--------------------------------------+
| Vous avez composé cet e-mail à une
| vitesse supérieure à x.xx cps. On ne
| peut normalement pas penser et écrire
| à une vitesse supérieure à y.yy cps, il
| est donc probable que votre réponse
| soit incohérente, mal conçue et/ou
| écrite sous le coup de l'émotion.
| Un minuteur vous empêchera d'envoyer
| un autre e-mail pendant l'heure qui suit.
|           [ANNULER]
+--------------------------------------+
```

La deuxième partie de mon souhait est plus émotionnelle. Évidemment, les anciens qui auraient dû calmer les débats sur le sujet sleep(1), malgré leurs années d'expérience sur le projet, n'ont jamais pris la peine de faire ce petit effort, alors pourquoi montent-ils sur leurs grands chevaux quand quelqu'un de beaucoup plus novice le fait à leur place ?

Si seulement je le savais.

Je sais par contre que la raison n'arrêtera pas un tel « conservatisme réactionnaire ». Il est possible que ces personnes soient frustrées par leur propre manque de contribution ces derniers temps, à moins d'avoir affaire à un cas aigu de « nous sommes vieux et grognons, NOUS savons comment les jeunes devraient se comporter ».

Dans un cas comme dans l'autre, c'est carrément contre productif mais je n'ai aucune suggestion à faire pour y remédier. Je pense que la meilleure chose à faire est de ne pas nourrir les trolls qui rodent dans les listes de diffusion : ignorez les, n'y répondez pas, oubliez leur existence.

J'espère que l'on pourra développer une base de contributeurs plus large et plus forte pour FreeBSD, et j'espère, qu'ensemble, nous pourrons éviter que les vieux grincheux et que les

[censuré] ne les broient, les recrachent et les fassent fuir avant même qu'ils n'aient un pied dans le projet.

À tous ceux qui traînent dans l'ombre, qui ne participent pas car trop effrayés par les gargouilles : je ne peux que vous présenter nos excuses et vous encourager à essayer malgré tout. Si ça ne tenait qu'à moi, l'ambiance dans le projet ne serait pas telle qu'elle est. [Poul-Henning]

ANNEXE D

Exemples d'instructions pour les rapports de bogues

Ceci est une version légèrement modifiée des instructions destinées aux nouveaux utilisateurs pour les rapports de bogues que l'on peut trouver sur le site du projet Subversion. Reportez-vous à la partie « Considérez tous les utilisateurs comme des volontaires potentiels » dans le chapitre 8 pour plus de détails sur l'importance de telles instructions. Vous pouvez retrouver ce document dans sa version originale à l'adresse http ://svn.collab.net/repos/svn/trunk/www/bugs.html.

Rapporter un bogue dans Subversion

Ce document vous indique comment et où rapporter les bogues (ce n'est pas une liste de bogues, vous pouvez la trouver ici par contre).

Où rapporter un bogue ?

– Si le bogue est dans Subversion lui-même, envoyez un e-mail à *users@subversion.tigris.org*. Une fois le bogue confirmé, quelqu'un, peut-être vous, peut l'inscrire dans le suivi des problèmes. (Ou, si vous êtes sûr de vous, envoyez le directement à notre liste de développement : *dev@subversion.tigris.org*. Mais si vous n'êtes pas certain, il vaut mieux l'envoyer à *users@* en premier, quelqu'un sera capable de vous dire si le comportement que vous avez rencontré est normal ou pas.)

- Si le bogue se trouve dans une librairie APR, rapportez-le s'il vous plaît à ces deux listes de diffusions : *dev@apr.apache.org* et *dev@subversion.tigris.org*.
- Si le bogue concerne la librairie Neon HTTP, rapportez-le s'il vous plaît à : *neon@webdav.org* et *dev@subversion.tigris.org*.
- Si le bogue est dans Apache HTTPD 2.0, rapportez-le s'il vous plaît à ces deux listes de diffusion : *dev@httpd.apache.org* et *dev@subversion.tigris.org*. La liste de diffusion Apache httpd génère beaucoup de trafic, il est donc possible que votre rapport de bogue passe inaperçu. Vous pouvez également enregistrer un rapport de bogue à l'adresse http ://httpd.apache.org/bug_report.html.
- *If the bug is in your rug, please give it a hug and keep it snug*. [Si l'insecte (bug) est sous votre tapis, faites lui un gros câlin et gardez le bien au chaud]

Comment rapporter un bogue ?

Assurez-vous déjà que c'est bien un bogue. Si Subversion ne réagit pas comme vous l'attendiez, vérifiez dans la documentation ou dans les archives des listes de diffusion des indices prouvant que c'est bien un comportement anormal. Bien sûr, si l'anomalie est évidente, par exemple si Subversion a détruit vos données ou fait fumer votre moniteur, vous pouvez vous fier à votre jugement. Mais en cas de doute n'hésitez pas à poser votre question en premier sur la liste de diffusion (*users@subversion.tigris.org*) ou sur IRC (irc.freenode.net) dans le canal #svn.

Lorsque vous serez certain que c'est un bogue, le mieux est d'en faire une description simple et de présenter une méthode de reproduction. Par exemple, si la première fois que vous avez découvert le bogue, il affectait cinq fichiers sur dix *commits* essayez de le reproduire avec seulement un fichier et un *commit*. Plus la méthode de reproduction est simple, plus il y a de chance qu'un développeur arrive à le reproduire et le corrige.

Lorsque vous rédigez la méthode de reproduction, ne vous contentez pas d'écrire une prose décrivant ce que vous faisiez et ce qui a déclenché le bogue. Il vaut mieux retranscrire exactement la série de commandes utilisées et leurs résultats. Servez-vous du copier/coller. Si des fichiers sont impliqués, prenez bien soin d'ajouter leur nom, voire leur contenu si vous pensez cela important. Le mieux que vous puissiez faire est de créer un script avec votre méthode de reproduction, cela nous aide beaucoup.

Au passage : vous êtes **sûr** d'utiliser la toute dernière version de Subversion, non ? :-) Il est possible que le bogue ait été corrigé, vous devriez tester votre méthode de reproduction avec l'arbre de développement de Subversion le plus récent.

En plus de la méthode de reproduction, nous aurons également besoin d'une description complète de l'environnement sous lequel vous avez reproduit le bogue. C'est-à-dire :

- votre système d'exploitation,
- la version et/ou la révision de Subversion,
- le compilateur et les options de configuration que vous avez utilisés pour compiler Subversion,
- toutes modifications que vous avez apportées à Subversion,
- la version de la Berkeley DB que vous utilisez avec Subversion, le cas échéant,
- tout autre chose qui pourrait être importante. Il vaut mieux recevoir trop d'informations que pas assez.

Une fois réunis tous ces éléments, vous serez prêt à rédiger le rapport. Commencez en donnant une description claire du bogue. C'est à dire que vous devez décrire le comportement attendu et le résultat obtenu. Même si le bogue peut vous sembler évident, il ne l'est pas forcément pour quelqu'un d'autre, il vaut donc mieux éviter de jouer aux devinettes. Continuez ensuite avec la description de votre environnement et la méthode de reproduction. Si vous voulez inclure des idées quant à la cause, ou même un correctif pour le bogue, c'est génial, voir http ://svn.collab.net/repos/svn/trunk/www/hacking.html #patches pour les instructions d'envoi de correctifs.

Transmettez toutes ces informations à *dev@subversion.tigris.org*, ou, si vous avez déjà contacté la liste de développeurs et qu'on vous a demandé de rapporter le problème, rendez-vous sur le système de référencement de problèmes et suivez les instructions. Merci. Nous savons qu'un bon rapport demande beaucoup de travail, mais c'est un gain de temps important pour nos développeurs et un grand pas vers la correction du bogue.

ANNEXE E

Copyright

sa mise à disposition. La meilleure manière de procéder est d'apposer un lien vers cette page Web. * Chacune de ces conditions peut être levée si vous obtenez l'autorisation du titulaire des droits sur cette oeuvre. * Rien dans ce contrat ne diminue ou ne restreint le droit moral du ou des auteurs.

Creative Commons Legal Code : Paternité - Partage Des Conditions Initiales à l'Identique 2.0

Creative Commons n'est pas un cabinet d'avocats et ne fournit pas de services de conseil juridique. La distribution de la présente version de ce contrat ne crée aucune relation juridique entre les parties du contrat présenté ci-après et Creative Commons. Creative Commons fournit cette offre de contrat-type en l'état, à seule fin d'information. Creative Commons ne saurait être tenu responsable des éventuels préjudices résultants du contenu ou de l'utilisation de ce contrat.

Contrat

L'Œuvre (telle que définie ci-dessous) est mise à disposition selon les termes du présent contrat appelé Contrat Public Creative Commons (dénommé ici « CPCC » ou « Contrat »). L'Œuvre est protégée par le droit de la propriété littéraire et artistique (droit d'auteur, droits voisins, droits des producteurs de bases de données) ou toute autre loi applicable. Toute utilisation de l'Œuvre autrement qu'explicitement autorisée selon ce Contrat ou le droit applicable est interdite.

L'exercice sur l'Œuvre de tout droit proposé par le présent contrat vaut acceptation de celui-ci. Selon les termes et les obligations du présent contrat, la partie Offrante propose à la partie Acceptante l'exercice de certains droits présentés ci-après, et l'Acceptant en approuve les termes et conditions d'utilisation.

1. Définitions

   a « Œuvre » : œuvre de l'esprit protégeable par le droit de la propriété littéraire et artistique ou toute loi applicable et qui est mise à disposition selon les termes du présent Contrat.

   b « Œuvre dite Collective » : une œuvre dans laquelle l'œuvre, dans sa forme intégrale et non modifiée, est assemblée en un ensemble collectif avec d'autres contributions qui constituent en elles-mêmes des œuvres séparées et indépendantes. Constituent notamment des Œuvres dites Collectives les publications périodiques, les anthologies ou les encyclopédies. Aux termes de la présente autorisation, une œuvre qui constitue une Œuvre dite Collective ne sera pas considérée comme une Œuvre dite Dérivée (telle que définie ci-après).

c « Œuvre dite Dérivée » : une œuvre créée soit à partir de l'Œuvre seule, soit à partir de l'Œuvre et d'autres œuvres préexistantes. Constituent notamment des Œuvres dites Dérivées les traductions, les arrangements musicaux, les adaptations théâtrales, littéraires ou cinématographiques, les enregistrements sonores, les reproductions par un art ou un procédé quelconque, les résumés, ou toute autre forme sous laquelle l'Œuvre peut être remaniée, modifiée, transformée ou adaptée, à l'exception d'une œuvre qui constitue une Œuvre dite Collective. Une Œuvre dite Collective ne sera pas considérée comme une Œuvre dite Dérivée aux termes du présent Contrat. Dans le cas où l'Œuvre serait une composition musicale ou un enregistrement sonore, la synchronisation de l'œuvre avec une image animée sera considérée comme une Œuvre dite Dérivée dans les propos de ce Contrat.

d « Auteur original » : la ou les personne(s) physique(s) ayant créé l'Œuvre.

e « Offrant » : la ou les personne(s) physique(s) ou morale(s) proposant la mise à disposition de l'Œuvre selon les termes du présent Contrat.

f « Acceptant » : la personne physique ou morale acceptant le présent contrat et exerçant des droits sans en avoir violé les termes au préalable ou ayant reçu l'autorisation expresse de l'Offrant d'exercer des droits dans le cadre du présent contrat malgré une précédente violation de ce contrat.

g « Options du Contrat » : les attributs génériques du Contrat tels qu'ils ont été choisis par l'Offrant et indiqués dans le titre de ce Contrat : Paternité - Pas d'Utilisation Commerciale - Partage Des Conditions Initiales A l'Identique.

2. Exceptions aux droits exclusifs. Aucune disposition de ce contrat n'a pour intention de réduire, limiter ou restreindre les prérogatives issues des exceptions aux droits, de l'épuisement des droits ou d'autres limitations aux droits exclusifs des ayants droit selon le droit de la propriété littéraire et artistique ou les autres lois applicables.

3. Autorisation. Soumis aux termes et conditions définis dans cette autorisation, et ceci pendant toute la durée de protection de l'Œuvre par le droit de la propriété littéraire et artistique ou le droit applicable, l'Offrant accorde à l'Acceptant l'autorisation mondiale d'exercer à titre gratuit et non-exclusif les droits suivants :

   a reproduire l'Œuvre, incorporer l'Œuvre dans une ou plusieurs Œuvres dites Collectives et reproduire l'Œuvre telle qu'incorporée dans lesdites Œuvres dites Collectives ;

   b créer et reproduire des Œuvres dites Dérivées ;

   c distribuer des exemplaires ou enregistrements, présenter, représenter ou communiquer l'Œuvre au public par tout procédé technique, y compris incorporée dans des Œuvres Collectives ;

   d distribuer des exemplaires ou phonogrammes, présenter, représenter ou communiquer au public des Œuvres dites Dérivées par tout procédé technique ;

   e lorsque l'Œuvre est une base de données, extraire et réutiliser des parties substantielles de l'Œuvre.

   Les droits mentionnés ci-dessus peuvent être exercés sur tous les supports, médias, procédés techniques et formats. Les droits ci-dessus incluent le droit d'effectuer les

modifications nécessaires techniquement à l'exercice des droits dans d'autres formats et procédés techniques. L'exercice de tous les droits qui ne sont pas expressément autorisés par l'Offrant ou dont il n'aurait pas la gestion demeure réservé, notamment les mécanismes de gestion collective obligatoire applicables décrits à l'article 4(d).

4. Restrictions. L'autorisation accordée par l'article 3 est expressément assujettie et limitée par le respect des restrictions suivantes :

   a L'Acceptant peut reproduire, distribuer, représenter ou communiquer au public l'Œuvre y compris par voie numérique uniquement selon les termes de ce Contrat. L'Acceptant doit inclure une copie ou l'adresse Internet (Identifiant Uniforme de Ressource) du présent Contrat à toute reproduction ou enregistrement de l'Œuvre que l'Acceptant distribue, représente ou communique au public y compris par voie numérique. L'Acceptant ne peut pas offrir ou imposer de conditions d'utilisation de l'Œuvre qui altèrent ou restreignent les termes du présent Contrat ou l'exercice des droits qui y sont accordés au bénéficiaire. L'Acceptant ne peut pas céder de droits sur l'Œuvre. L'Acceptant doit conserver intactes toutes les informations qui renvoient à ce Contrat et à l'exonération de responsabilité. L'Acceptant ne peut pas reproduire, distribuer, représenter ou communiquer au public l'Œuvre, y compris par voie numérique, en utilisant une mesure technique de contrôle d'accès ou de contrôle d'utilisation qui serait contradictoire avec les termes de cet Accord contractuel. Les mentions ci-dessus s'appliquent à l'Œuvre telle qu'incorporée dans une Œuvre dite Collective, mais, en dehors de l'Œuvre en elle-même, ne soumettent pas l'Œuvre dite Collective, aux termes du présent Contrat. Si l'Acceptant crée une Œuvre dite Collective, à la demande de tout Offrant, il devra, dans la mesure du possible, retirer de l'Œuvre dite Collective toute référence au dit Offrant, comme demandé. Si l'Acceptant crée une Œuvre dite Collective, à la demande de tout Auteur, il devra, dans la mesure du possible, retirer de l'Œuvre dite Collective toute référence au dit Auteur, comme demandé. Si l'Acceptant crée une Œuvre dite Dérivée, à la demande de tout Offrant, il devra, dans la mesure du possible, retirer de l'Œuvre dite Dérivée toute référence au dit Offrant, comme demandé. Si l'Acceptant crée une Œuvre dite Dérivée, à la demande de tout Auteur, il devra, dans la mesure du possible, retirer de l'Œuvre dite Dérivée toute référence au dit Auteur, comme demandé.

   b L'Acceptant peut reproduire, distribuer, représenter ou communiquer au public une Œuvre dite Dérivée y compris par voie numérique uniquement sous les termes de ce Contrat, ou d'une version ultérieure de ce Contrat comprenant les mêmes Options du Contrat que le présent Contrat, ou un Contrat Creative Commons iCommons comprenant les mêmes Options du Contrat que le présent Contrat (par exemple Paternité - Pas d'Utilisation Commerciale - Partage Des Conditions Initiales A l'Identique 2.0 Japon). L'Acceptant doit inclure une copie ou l'adresse Internet (Identifiant Uniforme de Ressource) du présent Contrat, ou d'un autre Contrat tel que décrit à la phrase précédente, à toute reproduction ou enregistrement de l'Œuvre dite Dérivée que l'Acceptant distribue, représente ou communique au public y compris par voie

numérique. L'Acceptant ne peut pas offrir ou imposer de conditions d'utilisation sur l'Œuvre dite Dérivée qui altèrent ou restreignent les termes du présent Contrat ou l'exercice des droits qui y sont accordés au bénéficiaire, et doit conserver intactes toutes les informations qui renvoient à ce Contrat et à l'avertissement sur les garanties. L'Acceptant ne peut pas reproduire, distribuer, représenter ou communiquer au public y compris par voie numérique l'Œuvre dite Dérivée en utilisant une mesure technique de contrôle d'accès ou de contrôle d'utilisation qui serait contradictoire avec les termes de cet Accord contractuel. Les mentions ci-dessus s'appliquent à l'Œuvre dite Dérivée telle qu'incorporée dans une Œuvre dite Collective, mais, en dehors de l'Œuvre dite Dérivée en elle-même, ne soumettent pas l'Œuvre Collective, aux termes du présent Contrat.

c Si l'Acceptant reproduit, distribue, représente ou communique au public, y compris par voie numérique, l'Œuvre ou toute Œuvre dite Dérivée ou toute Œuvre dite Collective, il doit conserver intactes toutes les informations sur le régime des droits et en attribuer la paternité à l'Auteur Original, de manière raisonnable au regard du médium ou du moyen utilisé. Il doit communiquer le nom de l'Auteur Original ou son éventuel pseudonyme s'il est indiqué ; le titre de l'Œuvre Originale s'il est indiqué ; dans la mesure du possible, l'adresse Internet ou Identifiant Uniforme de Ressource (URI), s'il existe, spécifié par l'Offrant comme associé à l'Œuvre, à moins que cette adresse ne renvoie pas aux informations légales (paternité et conditions d'utilisation de l'Œuvre). Dans le cas d'une Œuvre dite Dérivée, il doit indiquer les éléments identifiant l'utilisation l'Œuvre dans l'Œuvre dite Dérivée par exemple « Traduction anglaise de l'Œuvre par l'Auteur Original » ou « Scénario basé sur l'Œuvre par l'Auteur Original ». Ces obligations d'attribution de paternité doivent être exécutées de manière raisonnable. Cependant, dans le cas d'une Œuvre dite Dérivée ou d'une Œuvre dite Collective, ces informations doivent, au minimum, apparaître à la place et de manière aussi visible que celles à laquelle apparaissent les informations de même nature.

d Dans le cas où une utilisation de l'Œuvre serait soumise à un régime légal de gestion collective obligatoire, l'Offrant se réserve le droit exclusif de collecter ces redevances par l'intermédiaire de la société de perception et de répartition des droits compétente. Sont notamment concernés la radiodiffusion et la communication dans un lieu public de phonogrammes publiés à des fins de commerce, certains cas de retransmission par câble et satellite, la copie privée d'Œuvres fixées sur phonogrammes ou vidéogrammes, la reproduction par reprographie.

e Garantie et exonération de responsabilité

1. En mettant l'Œuvre à la disposition du public selon les termes de ce Contrat, l'Offrant déclare de bonne foi qu'à sa connaissance et dans les limites d'une enquête raisonnable :

   a L'Offrant a obtenu tous les droits sur l'Œuvre nécessaires pour pouvoir autoriser l'exercice des droits accordés par le présent

Contrat, et permettre la jouissance paisible et l'exercice licite de ces droits, ceci sans que l'Acceptant n'ait aucune obligation de verser de rémunération ou tout autre paiement ou droits, dans la limite des mécanismes de gestion collective obligatoire applicables décrits à l'article 4(e) ;

b L'Œuvre n'est constitutive ni d'une violation des droits de tiers, notamment du droit de la propriété littéraire et artistique, du droit des marques, du droit de l'information, du droit civil ou de tout autre droit, ni de diffamation, de violation de la vie privée ou de tout autre préjudice délictuel à l'égard de toute tierce partie.

2. A l'exception des situations expressément mentionnées dans le présent Contrat ou dans un autre accord écrit, ou exigées par la loi applicable, l'Œuvre est mise à disposition en l'état sans garantie d'aucune sorte, qu'elle soit expresse ou tacite, y compris à l'égard du contenu ou de l'exactitude de l'Œuvre.

f Limitation de responsabilité. A l'exception des garanties d'ordre public imposées par la loi applicable et des réparations imposées par le régime de la responsabilité vis-à-vis d'un tiers en raison de la violation des garanties prévues par l'article 5 du présent contrat, l'Offrant ne sera en aucun cas tenu responsable vis-à-vis de l'Acceptant, sur la base d'aucune théorie légale ni en raison d'aucun préjudice direct, indirect, matériel ou moral, résultant de l'exécution du présent Contrat ou de l'utilisation de l'Œuvre, y compris dans l'hypothèse où l'Offrant avait connaissance de la possible existence d'un tel préjudice.

g Résiliation

1. Tout manquement aux termes du contrat par l'Acceptant entraîne la résiliation automatique du Contrat et la fin des droits qui en découlent. Cependant, le contrat conserve ses effets envers les personnes physiques ou morales qui ont reçu de la part de l'Acceptant, en exécution du présent contrat, la mise à disposition d'Œuvres dites Dérivées, ou d'Œuvres dites Collectives, ceci tant qu'elles respectent pleinement leurs obligations. Les sections 1, 2, 5, 6 et 7 du contrat continuent à s'appliquer après la résiliation de celui-ci.
2. Dans les limites indiquées ci-dessus, le présent Contrat s'applique pendant toute la durée de protection de l'Œuvre selon le droit applicable. Néanmoins, l'Offrant se réserve à tout moment le droit d'exploiter l'Œuvre sous des conditions contractuelles différentes, ou d'en cesser la diffusion ; cependant, le recours à cette option ne doit pas conduire à retirer les effets du présent Contrat (ou de tout contrat qui a été ou doit être accordé selon les termes de ce Contrat), et ce Contrat continuera à s'appliquer dans tous ses effets jusqu'à ce que sa résiliation intervienne dans les conditions décrites ci-dessus.

h Divers

1. À chaque reproduction ou communication au public par voie numérique de l'Œuvre ou d'une Œuvre dite Collective par l'Acceptant, l'Offrant propose au bénéficiaire une offre de mise à disposition de l'Œuvre dans des termes et conditions identiques à ceux accordés à la partie Acceptante dans le présent Contrat.
2. À chaque reproduction ou communication au public par voie numérique d'une Œuvre dite Dérivée par l'Acceptant, l'Offrant propose au bénéficiaire une offre de mise à disposition du bénéficiaire de l'Œuvre originale dans des termes et conditions identiques à ceux accordés à la partie Acceptante dans le présent Contrat.
3. La nullité ou l'inapplicabilité d'une quelconque disposition de ce Contrat au regard de la loi applicable n'affecte pas celle des autres dispositions qui resteront pleinement valides et applicables. Sans action additionnelle par les parties à cet accord, lesdites dispositions devront être interprétées dans la mesure minimale nécessaire à leur validité et leur applicabilité.
4. Aucune limite, renonciation ou modification des termes ou dispositions du présent Contrat ne pourra être acceptée sans le consentement écrit et signé de la partie compétente.
5. Ce Contrat constitue le seul accord entre les parties à propos de l'Œuvre mise ici à disposition. Il n'existe aucun élément annexe, accord supplémentaire ou mandat portant sur cette Œuvre en dehors des éléments mentionnés ici. L'Offrant ne sera tenu par aucune disposition supplémentaire qui pourrait apparaître dans une quelconque communication en provenance de l'Acceptant. Ce Contrat ne peut être modifié sans l'accord mutuel écrit de l'Offrant et de l'Acceptant.
6. Le droit applicable est le droit français.

Creative Commons ne fait pas partie de ce Contrat et n'offre aucune forme de garantie relative à l'Œuvre. Creative Commons décline toute responsabilité à l'égard de l'Acceptant ou de toute autre partie, quel que soit le fondement légal de cette responsabilité et quel que soit le préjudice subi, direct, indirect, matériel ou moral, qui surviendrait en rapport avec le présent Contrat. Cependant, si Creative Commons s'est expressément identifié comme Offrant pour mettre une Œuvre à disposition selon les termes de ce Contrat, Creative Commons jouira de tous les droits et obligations d'un Offrant.

À l'exception des fins limitées à informer le public que l'Œuvre est mise à disposition sous CPCC, aucune des parties n'utilisera la marque « Creative Commons » ou toute autre indication ou logo afférent sans le consentement préalable écrit de Creative Commons. Toute utilisation autorisée devra être effectuée en conformité avec les lignes directrices de Creative Commons à jour au moment de l'utilisation, telles qu'elles sont disponibles sur son site Internet ou sur simple demande.

Creative Commons peut être contacté à http ://creativecommons.org/.