Geektionnerd : Impression 3D de disque

Source : [Impression 3D : une Américaine conçoit son propre disque vinyle (Numerama)]

Crédit : Simon Gee Giraudot (Creative Commons By-Sa)

Pour que le Libre aille vers l'Université (Libres conseils 6/42)

Université et communauté

Kevin Ottens

Kevin Ottens est un « hacker » de longue date au sein de la communauté KDE[1]. Il a largement contribué à la plateforme KDE, en particulier à la conception des API et frameworks. Diplômé en 2007, il est titulaire d'un doctorat en informatique qui l'a amené à travailler, en particulier, sur l'ingénierie des ontologies[2] et les systèmes multi-agents. Le travail de Kevin au KDAB inclut le développement de projets de recherche autour des technologies KDE. Il vit toujours à Toulouse, où il est enseignant à mi-temps dans son université d'origine.

Introduction

Les communautés du Libre sont principalement animées par l'effort de bénévoles. De plus, la plupart des personnes qui s'impliquent dans ces communautés le font lors de leur cursus universitaire. C'est la période idéale pour s'engager dans de telles aventures : on est jeune, plein d'énergie, curieux et l'on veut probablement façonner le monde à son image. Voilà tous les ingrédients d'un bon travail bénévole.

Mais, en même temps, être étudiant ne laisse pas forcément beaucoup de temps pour s'engager dans une communauté du Libre. En outre, la plupart de ces communautés sont assez vastes et les contacter peut faire peur.

Cela soulève évidement une question dérangeante : si les communautés du Libre ne réussissent pas à attirer la nouvelle génération de contributeurs talentueux, est-ce parce qu'elles

ne cherchent pas activement à étendre leur activité dans les universités ? Cette question pertinente, nous avons essayé d'y répondre dans le contexte d'une communauté qui produit des logiciels, à savoir KDE. Dans cet article, nous nous concentrerons sur les aspects auxquels nous n'avions pas initialement pensé mais qu'il nous a fallu aborder pour répondre à cette question.

Construire un partenariat avec une université locale

Tout commence, réellement, par le contact avec les étudiants eux-mêmes. Et pour ça, rien de mieux que de se rendre directement dans leur université et de leur montrer à quel point les communautés du Libre peuvent être accueillantes. À cet effet, nous avons construit un partenariat avec l'université *Paul Sabatier* de Toulouse plus précisément, avec l'un de ses cursus – nommé IUP ISI (NdT : Ingénierie des Systèmes Informatiques) – axé sur le développement logiciel.

L'IUP ISI était très orienté sur les connaissances « pratiques » et avait à ce titre un programme pré-établi pour les projets étudiants. Un point particulièrement intéressant de ce programme est le fait que les étudiants travaillent en équipe « inter-promotions ». Des étudiants de troisième et quatrième années, généralement en équipes de 7 à 10, apprennent à collaborer autour d'un objectif commun.

La première année de notre expérience, nous nous sommes raccrochés à ce programme en proposant de nouveaux sujets pour les projets, et en nous concentrant sur des logiciels développés au sein de la communauté KDE. Henri Massie, directeur du cursus, a très bien accueilli cette idée, et nous a laissés mettre cette expérience en place. Pour cette première année, nous nous sommes vu attribuer deux créneaux horaires pour les « projets KDE ».

Pour créer rapidement un climat de confiance, nous avons décidé, cette année-là, d'offrir quelques garanties concernant le travail des étudiants :

- Aider les professeurs à avoir confiance dans les sujets abordés : les projets sélectionnés étaient très proches des sujets enseignés à l'IUP ISI (c'est pourquoi, pour cette année, nous avons ciblé un outil de modélisation UML et un outil de gestion de projet) ;
- donner un maximum de visibilité aux professeurs : nous avons mis à leur disposition un serveur, sur lequel étaient régulièrement compilés les projets étudiants, accessible à distance à des fins de test ;
- faciliter la participation des étudiants à la communauté : les responsables des projets étaient désignés pour jouer le rôle du « client », soumettre leurs exigences aux étudiants et les aider à trouver leur chemin dans le dédale de la communauté ;
- enfin, pour mettre le pied à l'étrier aux étudiants, nous leur avons donné un petit cours sur « Comment développer avec Qt et les autres frameworks produits par KDE ».

Au moment de l'écriture de ces pages, cela fait cinq ans que nous menons de tels projets. De petits ajustements dans l'organisation ont été apportés ici et là, mais la plupart des idées sous-jacentes sont restées les mêmes. La majorité des changements ont été le résultat d'un intérêt grandissant de la communauté, désireuse d'établir un partenariat avec les étudiants, et d'une plus grande liberté pour nous quant aux sujets que nous pourrions couvrir dans nos projets.

De plus, tout au long de ces années, le directeur nous a donné une aide et des encouragements constants, attribuant effectivement plus de créneaux pour les projets de la communauté du Libre et prouvant que notre stratégie d'intégration était juste : établir de la confiance dès le début est la clé d'un partenariat entre la communauté du Libre

et l’Université.

Comprendre que l’enseignement est un processus interactif

Durant ces années à tisser des liens entre la communauté KDE et la filière IUP ISI, nous nous sommes retrouvés en situation d’enseignement afin d’assister les étudiants dans des tâches liées à leurs projets. Quand vous n’avez jamais enseigné à une classe pleine d’étudiants, vous avez sans doute encore une image de vous, il y a quelques années, assis dans une classe. En effet, la plupart des enseignants ont un jour été étudiants… Parfois même, pas le genre d’étudiant très discipliné ni attentif. Vous aviez sans doute l’impression d’être submergé : l’enseignant entrait dans la salle, faisait face aux étudiants, et déversait sur vous ses connaissances.

Ce stéréotype est ce que la plupart gardent à l’esprit de leurs années d’études et la première fois qu’ils se retrouvent en situation d’enseigner, ils veulent reproduire ce stéréotype : arriver avec un savoir à transmettre.

La bonne nouvelle, c’est que rien n’est plus éloigné de la vérité que ce stéréotype. La mauvaise nouvelle, c’est que si vous essayez de reproduire ce stéréotype, vous allez très probablement faire fuir vos étudiants et ne ferez face qu’à un manque de motivation pour participer à la communauté. L’image que vous donnez de vous est la toute première chose dont ils vont se souvenir de la communauté : la première fois que vous entrez dans la salle de classe, vous êtes, pour eux, la communauté.

Pour éviter de tomber dans le piège de ce stéréotype, il vous faut prendre un peu de recul et réaliser ce que signifie réellement d’enseigner. Ce n’est pas un processus à sens unique où l’on livre la connaissance aux étudiants. Nous sommes arrivés à la conclusion que c’est en fait un processus

à double sens : vous êtes amené à créer une relation symbiotique avec vos étudiants. Les étudiants et les enseignants doivent tous sortir de la salle de classe avec de nouvelles connaissances. Il vous faut livrer votre expertise, bien sûr, mais afin de le faire efficacement, vous devez en permanence vous adapter au cadre de référence de vos étudiants. C'est un travail qui rend très humble.

Cette prise de conscience génère pas mal de changements dans la manière d'entreprendre votre enseignement.

- Vous allez devoir comprendre la culture de vos étudiants. Ils ont probablement des expériences assez différentes des vôtres et vous allez devoir adapter votre discours à eux ; par exemple, les étudiants que nous avons formés font tous partie de la fameuse « génération Y » qui, en matière de leadership, loyauté et confiance, présente des caractéristiques assez différentes de la génération précédente.
- Vous allez devoir réévaluer votre propre expertise, puisque vous allez devoir adapter votre discours à leur culture. Vous aborderez vos propres connaissances selon un angle très différent de celui dont vous avez l'habitude, ce qui vous mènera inévitablement à des découvertes dans des domaines que vous pensiez maîtriser.
- Enfin, vous allez devoir vous forger des compétences en présentation ; l'enseignement consiste réellement à sortir de votre zone de confort afin de présenter vos propres connaissances tout en les gardant intéressantes et divertissantes pour votre audience. Cela fera de vous un meilleur présentateur.

Ainsi, vous deviendrez un meilleur enseignant. De plus, vous remplirez mieux vos objectifs : des étudiants bien formés, dont certains s'engageront dans la communauté du Libre.

Conclusion

Au bout du compte, pourquoi feriez-vous tous ces efforts pour établir une relation de confiance avec une université et sortir de votre zone de confort en améliorant votre manière d'enseigner ? Eh bien, cela se résume vraiment à la question initiale à laquelle nous avons tenté de répondre :

Si les communautés du Libre ne réussissent pas à attirer de nouveaux contributeurs venus des universités, est-ce simplement dû à leur inaction ?

D'après notre expérience, la réponse est oui. Au cours de ces cinq années passées à bâtir un partenariat avec l'IUP ISI, nous avons attiré deux étudiants par année en moyenne. Certains d'entre eux nous ont quittés après quelque temps, mais certains sont devenus des contributeurs très actifs. Les autres gardent encore une certaine nostalgie de cette période de leur vie et continuent de nous soutenir même s'ils ne contribuent pas directement. En ce moment même, nous avons une équipe locale KDE qui a réussi à organiser efficacement une conférence de deux jours pour notre dernière « release party » (NdT : soirée de lancement).

Parmi ces anciens étudiants, pas un seul ne se serait impliqué dans le projet KDE sans ces projets universitaires. Nous serions passés complètement à côté de ces talents. Par chance, nous n'avons pas été inactifs.

[1] http://www.kde.org

[2] https://fr.wikipedia.org/wiki/Ontologie_(informatique)

Université et communauté

Kevin Ottens

Kevin Ottens est un « hacker » de longue date au sein de la communauté KDE[1]. Il a largement contribué à la plateforme KDE, en particulier à la conception des API et frameworks. Diplômé en 2007, il est titulaire d'un doctorat en informatique qui l'a amené à travailler, en particulier, sur l'ingénierie des ontologies[2] et les systèmes multi-agents. Le travail de Kevin au KDAB inclut le développement de projets de recherche autour des technologies KDE. Il vit toujours à Toulouse, où il est enseignant à mi-temps dans son université d'origine.

Introduction

Les communautés du Libre sont principalement animées par l'effort de bénévoles. De plus, la plupart des personnes qui s'impliquent dans ces communautés le font lors de leur cursus universitaire. C'est la période idéale pour s'engager dans de telles aventures : on est jeune, plein d'énergie, curieux et

l’on veut probablement façonner le monde à son image. Voilà tous les ingrédients d’un bon travail bénévole.

Mais, en même temps, être étudiant ne laisse pas forcément beaucoup de temps pour s’engager dans une communauté du Libre. En outre, la plupart de ces communautés sont assez vastes et les contacter peut faire peur.

Cela soulève évidement une question dérangeante : si les communautés du Libre ne réussissent pas à attirer la nouvelle génération de contributeurs talentueux, est-ce parce qu’elles ne cherchent pas activement à étendre leur activité dans les universités ? Cette question pertinente, nous avons essayé d’y répondre dans le contexte d’une communauté qui produit des logiciels, à savoir KDE. Dans cet article, nous nous concentrerons sur les aspects auxquels nous n’avions pas initialement pensé mais qu’il nous a fallu aborder pour répondre à cette question.

Construire un partenariat avec une université locale

Tout commence, réellement, par le contact avec les étudiants eux-mêmes. Et pour ça, rien de mieux que de se rendre directement dans leur université et de leur montrer à quel point les communautés du Libre peuvent être accueillantes. À cet effet, nous avons construit un partenariat avec l’université *Paul Sabatier* de Toulouse plus précisément, avec l’un de ses cursus – nommé IUP ISI (NdT : Ingénierie des Systèmes Informatiques) – axé sur le développement logiciel.

L’IUP ISI était très orienté sur les connaissances « pratiques » et avait à ce titre un programme pré-établi pour les projets étudiants. Un point particulièrement intéressant de ce programme est le fait que les étudiants travaillent en équipe « inter-promotions ». Des étudiants de troisième et quatrième années, généralement en équipes de 7 à 10,

apprennent à collaborer autour d'un objectif commun.

La première année de notre expérience, nous nous sommes raccrochés à ce programme en proposant de nouveaux sujets pour les projets, et en nous concentrant sur des logiciels développés au sein de la communauté KDE. Henri Massie, directeur du cursus, a très bien accueilli cette idée, et nous a laissés mettre cette expérience en place. Pour cette première année, nous nous sommes vu attribuer deux créneaux horaires pour les « projets KDE ».

Pour créer rapidement un climat de confiance, nous avons décidé, cette année-là, d'offrir quelques garanties concernant le travail des étudiants :

- Aider les professeurs à avoir confiance dans les sujets abordés : les projets sélectionnés étaient très proches des sujets enseignés à l'IUP ISI (c'est pourquoi, pour cette année, nous avons ciblé un outil de modélisation UML et un outil de gestion de projet) ;
- donner un maximum de visibilité aux professeurs : nous avons mis à leur disposition un serveur, sur lequel étaient régulièrement compilés les projets étudiants, accessible à distance à des fins de test ;
- faciliter la participation des étudiants à la communauté : les responsables des projets étaient désignés pour jouer le rôle du « client », soumettre leurs exigences aux étudiants et les aider à trouver leur chemin dans le dédale de la communauté ;
- enfin, pour mettre le pied à l'étrier aux étudiants, nous leur avons donné un petit cours sur « Comment développer avec Qt et les autres frameworks produits par KDE ».

Au moment de l'écriture de ces pages, cela fait cinq ans que nous menons de tels projets. De petits ajustements dans l'organisation ont été apportés ici et là, mais la plupart des idées sous-jacentes sont restées les mêmes. La majorité des

changements ont été le résultat d'un intérêt grandissant de la communauté, désireuse d'établir un partenariat avec les étudiants, et d'une plus grande liberté pour nous quant aux sujets que nous pourrions couvrir dans nos projets.

De plus, tout au long de ces années, le directeur nous a donné une aide et des encouragements constants, attribuant effectivement plus de créneaux pour les projets de la communauté du Libre et prouvant que notre stratégie d'intégration était juste : établir de la confiance dès le début est la clé d'un partenariat entre la communauté du Libre et l'Université.

Comprendre que l'enseignement est un processus interactif

Durant ces années à tisser des liens entre la communauté KDE et la filière IUP ISI, nous nous sommes retrouvés en situation d'enseignement afin d'assister les étudiants dans des tâches liées à leurs projets. Quand vous n'avez jamais enseigné à une classe pleine d'étudiants, vous avez sans doute encore une image de vous, il y a quelques années, assis dans une classe. En effet, la plupart des enseignants ont un jour été étudiants… Parfois même, pas le genre d'étudiant très discipliné ni attentif. Vous aviez sans doute l'impression d'être submergé : l'enseignant entrait dans la salle, faisait face aux étudiants, et déversait sur vous ses connaissances.

Ce stéréotype est ce que la plupart gardent à l'esprit de leurs années d'études et la première fois qu'ils se retrouvent en situation d'enseigner, ils veulent reproduire ce stéréotype : arriver avec un savoir à transmettre.

La bonne nouvelle, c'est que rien n'est plus éloigné de la vérité que ce stéréotype. La mauvaise nouvelle, c'est que si vous essayez de reproduire ce stéréotype, vous allez très probablement faire fuir vos étudiants et ne ferez face qu'à un

manque de motivation pour participer à la communauté. L'image que vous donnez de vous est la toute première chose dont ils vont se souvenir de la communauté : la première fois que vous entrez dans la salle de classe, vous êtes, pour eux, la communauté.

Pour éviter de tomber dans le piège de ce stéréotype, il vous faut prendre un peu de recul et réaliser ce que signifie réellement d'enseigner. Ce n'est pas un processus à sens unique où l'on livre la connaissance aux étudiants. Nous sommes arrivés à la conclusion que c'est en fait un processus à double sens : vous êtes amené à créer une relation symbiotique avec vos étudiants. Les étudiants et les enseignants doivent tous sortir de la salle de classe avec de nouvelles connaissances. Il vous faut livrer votre expertise, bien sûr, mais afin de le faire efficacement, vous devez en permanence vous adapter au cadre de référence de vos étudiants. C'est un travail qui rend très humble.

Cette prise de conscience génère pas mal de changements dans la manière d'entreprendre votre enseignement.

- Vous allez devoir comprendre la culture de vos étudiants. Ils ont probablement des expériences assez différentes des vôtres et vous allez devoir adapter votre discours à eux ; par exemple, les étudiants que nous avons formés font tous partie de la fameuse « génération Y » qui, en matière de leadership, loyauté et confiance, présente des caractéristiques assez différentes de la génération précédente.
- Vous allez devoir réévaluer votre propre expertise, puisque vous allez devoir adapter votre discours à leur culture. Vous aborderez vos propres connaissances selon un angle très différent de celui dont vous avez l'habitude, ce qui vous mènera inévitablement à des découvertes dans des domaines que vous pensiez maîtriser.
- Enfin, vous allez devoir vous forger des compétences en

présentation ; l’enseignement consiste réellement à sortir de votre zone de confort afin de présenter vos propres connaissances tout en les gardant intéressantes et divertissantes pour votre audience. Cela fera de vous un meilleur présentateur.

Ainsi, vous deviendrez un meilleur enseignant. De plus, vous remplirez mieux vos objectifs : des étudiants bien formés, dont certains s’engageront dans la communauté du Libre.

Conclusion

Au bout du compte, pourquoi feriez-vous tous ces efforts pour établir une relation de confiance avec une université et sortir de votre zone de confort en améliorant votre manière d’enseigner ? Eh bien, cela se résume vraiment à la question initiale à laquelle nous avons tenté de répondre :

Si les communautés du Libre ne réussissent pas à attirer de nouveaux contributeurs venus des universités, est-ce simplement dû à leur inaction ?

D’après notre expérience, la réponse est oui. Au cours de ces cinq années passées à bâtir un partenariat avec l’IUP ISI, nous avons attiré deux étudiants par année en moyenne. Certains d’entre eux nous ont quittés après quelque temps, mais certains sont devenus des contributeurs très actifs. Les autres gardent encore une certaine nostalgie de cette période de leur vie et continuent de nous soutenir même s’ils ne contribuent pas directement. En ce moment même, nous avons une équipe locale KDE qui a réussi à organiser efficacement une conférence de deux jours pour notre dernière « release party » (NdT : soirée de lancement).

Parmi ces anciens étudiants, pas un seul ne se serait impliqué dans le projet KDE sans ces projets universitaires. Nous serions passés complètement à côté de ces talents. Par chance, nous n’avons pas été inactifs.

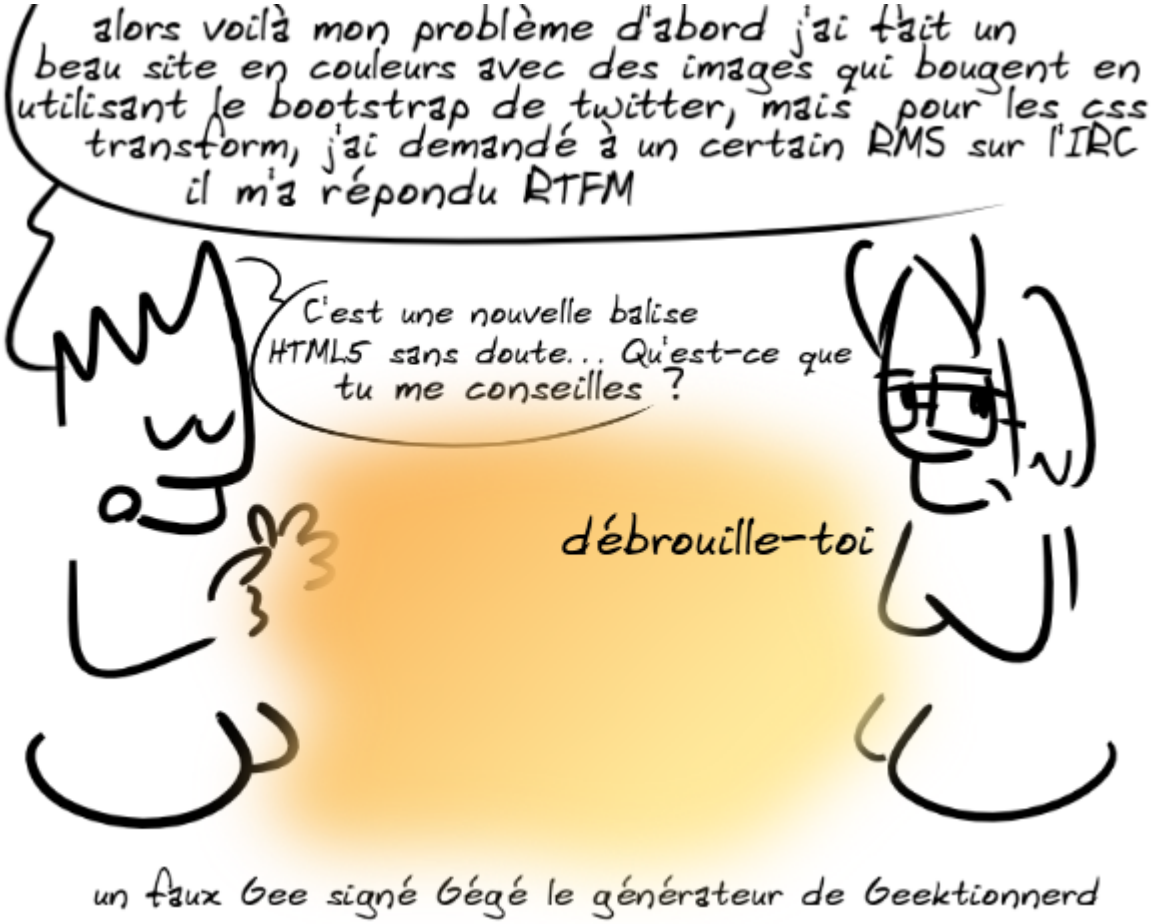
[1] http://www.kde.org

[2] https://fr.wikipedia.org/wiki/Ontologie_(informatique)

Du bon usage des mentors (Libres conseils 5/42)

Vous finirez par savoir tout ce qu'ils ont oublié

Leslie Hawthorn



Gestionnaire de communautés internationalement reconnue, conférencière et auteur, Leslie Hawthorn a plus de 10 ans d'expérience dans la gestion de projets high tech, le marketing et les relations publiques. Elle a récemment rejoint AppFog[(1)] en tant que responsable de la communauté, où elle est chargée du recrutement de développeurs. Auparavant, elle a travaillé comme responsable de communication au laboratoire open source de l'Université de l'état de l'Oregon et comme responsable de programme au sein de l'équipe open source de Google, où elle a géré le Google Summer of Code[(2)], créé le concours que l'on connaît maintenant sous le nom Google Code-in et lancé le blog de développement open source de la société.

> *« La documentation la plus importante pour les nouveaux utilisateurs concerne les bases : comment mettre rapidement 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. » Karl Fogel, Produire du logiciel libre[(3)]*

Quand pour la première fois vous commencez à travailler sur un projet de logiciel libre et *open source*, la courbe d'apprentissage est raide et le chemin difficile. Vous risquez de vous retrouver abonné à des listes de diffusion ou dans des salons de discussion avec toutes sortes de gens renommés, comme le créateur de votre langage de programmation favori ou le responsable de votre logiciel préféré, et vous vous demanderez si vous serez un jour suffisamment qualifié pour contribuer efficacement. Ce dont vous n'aurez pas forcément conscience, c'est à quel point ces gens sages ont oublié le

long chemin qui les a menés au succès.

Prenons une analogie simple : dans un projet *open source*, le processus d'apprentissage, comme utilisateur ou comme développeur, c'est un peu comme apprendre à faire du vélo. Pour les cyclistes expérimentés, « c'est aussi facile que de monter à vélo ». Vous avez probablement fait du vélo quelques fois et vous comprenez son architecture : une selle, des roues, des pédales et un guidon. Pourtant, vous montez en selle, concentré sur votre avancée et soudainement vous découvrez que ce n'est pas aussi simple que ce que vous pensiez : à quelle hauteur faut-il régler votre selle ? Quel équipement vous faut-il quand vous grimpez une colline ? Quand vous en descendez une ? D'ailleurs, avez-vous vraiment besoin de ce casque ? (un conseil : oui, absolument).

Lorsque vous vous mettez au vélo, vous ne savez même pas quelles questions poser et vous ne les trouverez que dans vos genoux endoloris, des points de côté et des courbatures dans le dos. Même dans ce cas, vos questions ne correspondront pas toujours aux réponses dont vous avez besoin ; quelqu'un pourrait s'aviser de vous dire d'abaisser la selle quand vous lui dites que vos genoux font mal, mais d'autres peuvent aussi bien supposer que tout ça est nouveau pour vous et que vous finirez bien par le découvrir par vous-même. Ils ont oublié qu'il faut se battre avec les changements de vitesse, se rendre compte qu'on n'a pas les bons éclairages ni les réflecteurs adéquats, comment tourner à gauche en levant la bonne main, parce qu'ils font du vélo depuis si longtemps que tous ces gestes sont pour eux comme une seconde nature.

Le scénario reste le même lorsque vous débutez dans le monde des logiciels libres et *open source*. Lorsque vous compilez un paquet pour la première fois, vous allez inévitablement arriver à un obscur message d'erreur ou un autre genre d'échec. Et lorsque vous demanderez de l'aide, une bonne âme vous dira sans doute : « c'est facile, il suffit de faire make -toto -titi -tata ». Sauf que pour vous, ce n'est pas facile.

Il n’y aura probablement pas de documentation pour toto, titi ne fera pas ce qu’il est supposé faire et qu’est ce que ce truc tata avec ses huit homonymes sur Wikipédia ? Évidemment, vous ne voulez pas être un boulet, mais vous allez avoir besoin d’aide pour réussir vraiment à faire quelque chose.

Vous allez peut-être persister à reprendre les mêmes étapes, rencontrer les mêmes échecs, et la frustration ira grandissant. Peut-être que vous allez vous lever pour prendre un café en pensant que vous reviendrez sur le problème plus tard. Ce qu’aucun de nous dans le monde des logiciels libres et *open source* ne voudrait voir se produire, c’est précisément ce qui se passe pour beaucoup : boire cette tasse de café est infiniment meilleur que de se sentir ignorant et intimidé, et vous n’allez pas plus avant dans votre découverte du Libre.

Prenez conscience dès maintenant que vous finirez par connaître ces choses que les experts autour de vous ont oubliées ou qu’ils ne communiquent pas car ces étapes sont évidentes pour eux. Toute personne plus expérimentée que vous est passée par les mêmes affres que vous en ce moment pour apprendre à faire ce que vous vous efforcez de faire. Voici quelques conseils pour rendre votre parcours plus facile :

N’attendez pas trop longtemps avant de demander de l’aide. Personne ne veut être un boulet et personne n’aime avoir l’air perdu. Cela dit, si vous n’arrivez pas à résoudre votre problème après avoir essayé pendant 15 minutes, il est temps de demander de l’aide. Vérifiez dans la documentation sur le site web du projet que vous utilisez le bon canal IRC, le forum ou la liste de diffusion pour demander de l’aide. De nombreux projets ont des canaux d’aide en ligne spécialement pour les débutants, gardez donc un œil sur des mots tels que mentor, débutant et mise en route.

Parlez de votre processus (de réflexion). Il ne s’agit pas seulement de poser des questions, mais de savoir quelles sont les bonnes questions à poser. Au début, vous ne saurez pas

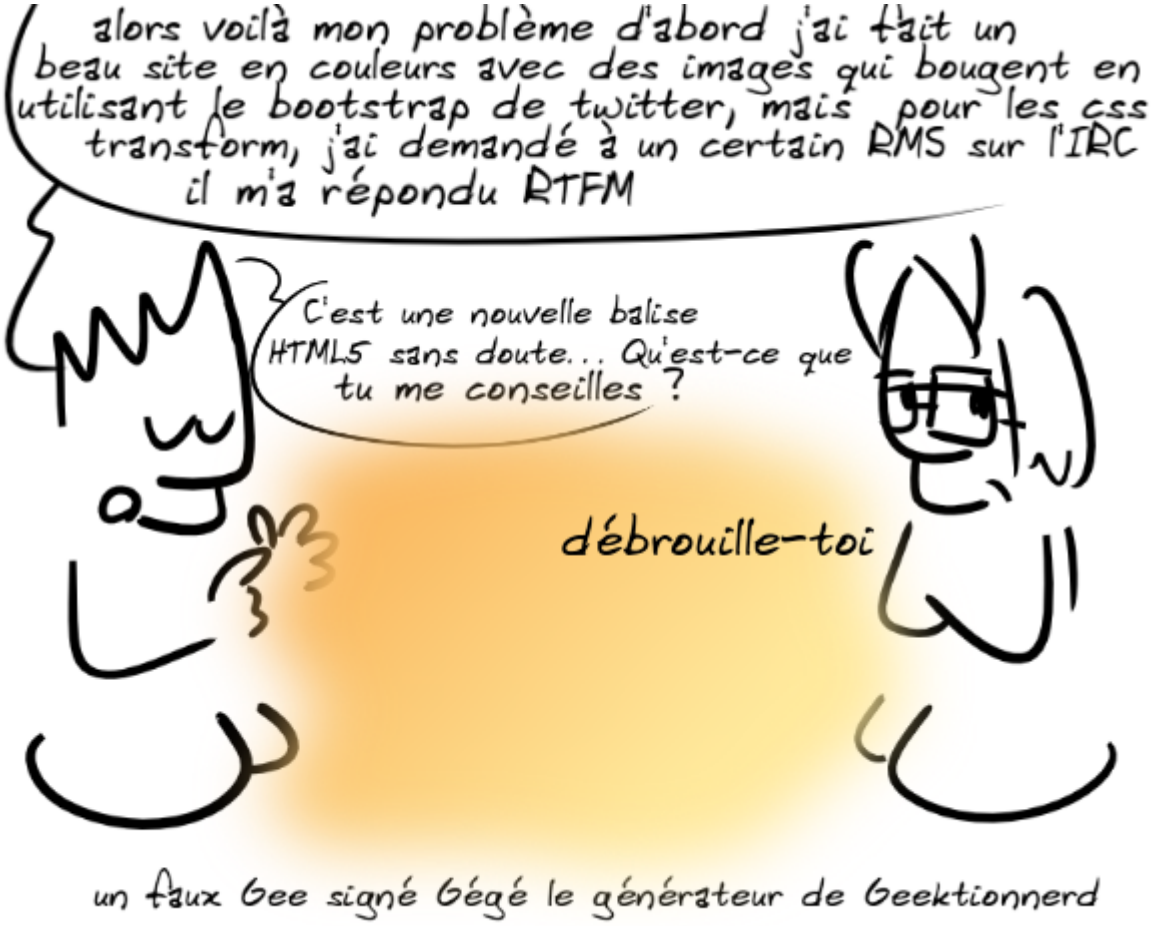
forcément quelles sont ces bonnes questions. Donc quand vous demanderez de l'aide, détaillez ce que vous essayez de faire, les étapes par lesquelles vous êtes passé, et les problèmes que vous avez rencontrés. Signalez aux futurs mentors du canal IRC ou de la liste de diffusion que vous avez lu le manuel en incluant des liens vers la documentation que vous avez lue sur le sujet. Si vous n'avez trouvé aucune documentation, le signaler poliment peut aider.

Apprenez à connaître votre propre valeur. En tant que nouveau contributeur dans un projet, vous êtes un atout précieux. Non pas pour vos connaissances, mais pour votre ignorance. Lorsque vous commencez à travailler sur des logiciels libres et *open source*, rien n'est assez évident à vos yeux et tout mérite donc d'être expliqué. Prenez des notes à propos des problèmes que vous avez rencontrés, et de la façon dont ils ont été résolus. Puis utilisez ces notes pour mettre à jour la documentation du projet, travailler avec la communauté à des démos vidéo ou autres documents de formation pour les cas les plus épineux. Quand vous rencontrez un problème vraiment frustrant, comprenez que vous êtes en position idéale pour faire en sorte que le prochain qui tombera dessus ne rencontrera pas les mêmes difficultés.

1. http://www.appfog.com/ ^
2. http://fr.wikipedia.org/wiki/Google_Summer_of_Code ^
3. http://framabook.org/8-produire-du-logiciel-libre ^

Vous finirez par savoir tout ce qu'ils ont oublié

Leslie Hawthorn



Gestionnaire de communautés internationalement reconnue, conférencière et auteur, Leslie Hawthorn a plus de 10 ans d'expérience dans la gestion de projets high tech, le marketing et les relations publiques. Elle a récemment rejoint AppFog[(1)] en tant que responsable de la communauté, où elle est chargée du recrutement de développeurs. Auparavant, elle a travaillé comme responsable de communication au laboratoire open source de l'Université de l'état de l'Oregon et comme responsable de programme au sein de l'équipe open source de Google, où elle a géré le Google Summer of Code[(2)], créé le concours que l'on connaît maintenant sous le nom Google Code-in et lancé le blog de développement open source de la société.

> *« La documentation la plus importante pour les nouveaux utilisateurs concerne les bases : comment mettre rapidement 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. » Karl Fogel, Produire du logiciel libre*[(3)]

Quand pour la première fois vous commencez à travailler sur un projet de logiciel libre et *open source*, la courbe d'apprentissage est raide et le chemin difficile. Vous risquez de vous retrouver abonné à des listes de diffusion ou dans des salons de discussion avec toutes sortes de gens renommés, comme le créateur de votre langage de programmation favori ou le responsable de votre logiciel préféré, et vous vous demanderez si vous serez un jour suffisamment qualifié pour contribuer efficacement. Ce dont vous n'aurez pas forcément conscience, c'est à quel point ces gens sages ont oublié le long chemin qui les a menés au succès.

Prenons une analogie simple : dans un projet *open source*, le processus d'apprentissage, comme utilisateur ou comme développeur, c'est un peu comme apprendre à faire du vélo. Pour les cyclistes expérimentés, « c'est aussi facile que de monter à vélo ». Vous avez probablement fait du vélo quelques fois et vous comprenez son architecture : une selle, des roues, des pédales et un guidon. Pourtant, vous montez en selle, concentré sur votre avancée et soudainement vous découvrez que ce n'est pas aussi simple que ce que vous pensiez : à quelle hauteur faut-il régler votre selle ? Quel équipement vous faut-il quand vous grimpez une colline ? Quand vous en descendez une ? D'ailleurs, avez-vous vraiment besoin de ce casque ? (un conseil : oui, absolument).

Lorsque vous vous mettez au vélo, vous ne savez même pas quelles questions poser et vous ne les trouverez que dans vos genoux endoloris, des points de côté et des courbatures dans

le dos. Même dans ce cas, vos questions ne correspondront pas toujours aux réponses dont vous avez besoin ; quelqu'un pourrait s'aviser de vous dire d'abaisser la selle quand vous lui dites que vos genoux font mal, mais d'autres peuvent aussi bien supposer que tout ça est nouveau pour vous et que vous finirez bien par le découvrir par vous-même. Ils ont oublié qu'il faut se battre avec les changements de vitesse, se rendre compte qu'on n'a pas les bons éclairages ni les réflecteurs adéquats, comment tourner à gauche en levant la bonne main, parce qu'ils font du vélo depuis si longtemps que tous ces gestes sont pour eux comme une seconde nature.

Le scénario reste le même lorsque vous débutez dans le monde des logiciels libres et *open source*. Lorsque vous compilez un paquet pour la première fois, vous allez inévitablement arriver à un obscur message d'erreur ou un autre genre d'échec. Et lorsque vous demanderez de l'aide, une bonne âme vous dira sans doute : « c'est facile, il suffit de faire make -toto -titi -tata ». Sauf que pour vous, ce n'est pas facile. Il n'y aura probablement pas de documentation pour toto, titi ne fera pas ce qu'il est supposé faire et qu'est ce que ce truc tata avec ses huit homonymes sur Wikipédia ? Évidemment, vous ne voulez pas être un boulet, mais vous allez avoir besoin d'aide pour réussir vraiment à faire quelque chose.

Vous allez peut-être persister à reprendre les mêmes étapes, rencontrer les mêmes échecs, et la frustration ira grandissant. Peut-être que vous allez vous lever pour prendre un café en pensant que vous reviendrez sur le problème plus tard. Ce qu'aucun de nous dans le monde des logiciels libres et *open source* ne voudrait voir se produire, c'est précisément ce qui se passe pour beaucoup : boire cette tasse de café est infiniment meilleur que de se sentir ignorant et intimidé, et vous n'allez pas plus avant dans votre découverte du Libre.

Prenez conscience dès maintenant que vous finirez par connaître ces choses que les experts autour de vous ont oubliées ou qu'ils ne communiquent pas car ces étapes sont

évidentes pour eux. Toute personne plus expérimentée que vous est passée par les mêmes affres que vous en ce moment pour apprendre à faire ce que vous vous efforcez de faire. Voici quelques conseils pour rendre votre parcours plus facile :

N'attendez pas trop longtemps avant de demander de l'aide. Personne ne veut être un boulet et personne n'aime avoir l'air perdu. Cela dit, si vous n'arrivez pas à résoudre votre problème après avoir essayé pendant 15 minutes, il est temps de demander de l'aide. Vérifiez dans la documentation sur le site web du projet que vous utilisez le bon canal IRC, le forum ou la liste de diffusion pour demander de l'aide. De nombreux projets ont des canaux d'aide en ligne spécialement pour les débutants, gardez donc un œil sur des mots tels que mentor, débutant et mise en route.

Parlez de votre processus (de réflexion). Il ne s'agit pas seulement de poser des questions, mais de savoir quelles sont les bonnes questions à poser. Au début, vous ne saurez pas forcément quelles sont ces bonnes questions. Donc quand vous demanderez de l'aide, détaillez ce que vous essayez de faire, les étapes par lesquelles vous êtes passé, et les problèmes que vous avez rencontrés. Signalez aux futurs mentors du canal IRC ou de la liste de diffusion que vous avez lu le manuel en incluant des liens vers la documentation que vous avez lue sur le sujet. Si vous n'avez trouvé aucune documentation, le signaler poliment peut aider.

Apprenez à connaître votre propre valeur. En tant que nouveau contributeur dans un projet, vous êtes un atout précieux. Non pas pour vos connaissances, mais pour votre ignorance. Lorsque vous commencez à travailler sur des logiciels libres et *open source*, rien n'est assez évident à vos yeux et tout mérite donc d'être expliqué. Prenez des notes à propos des problèmes que vous avez rencontrés, et de la façon dont ils ont été résolus. Puis utilisez ces notes pour mettre à jour la documentation du projet, travailler avec la communauté à des démos vidéo ou autres documents de formation pour les cas les

plus épineux. Quand vous rencontrez un problème vraiment frustrant, comprenez que vous êtes en position idéale pour faire en sorte que le prochain qui tombera dessus ne rencontrera pas les mêmes difficultés.

1. http://www.appfog.com/ ↩
2. http://fr.wikipedia.org/wiki/Google_Summer_of_Code ↩
3. http://framabook.org/8-produire-du-logiciel-libre ↩

Prévoir l'avenir d'un projet open source (Libres conseils 4/42)

Traduction Framalang : ga3lig, Coco, Aa, Lignusta, goofy, jcr83, peupleLa (relectures), Sylvain, CoudCoud, lamessen + Julius22

Préparez-vous pour le futur : l'évolution des équipes dans le logiciel libre et *open source*

par Felipe Ortega

Felipe Ortega est chercheur et chef de projet à Libresoft, un groupe de recherche de l'Université Rey Juan Carlos [1] *en Espagne. Il développe de nouvelles méthodologies pour analyser les communautés collaboratives ouvertes (comme les projets de logiciels libres, Wikipédia et les réseaux sociaux). Il a mené des recherches approfondies sur le projet Wikipédia et sa communauté de contributeurs. Felipe participe activement à la recherche, la promotion et l'éducation/formation sur le logiciel libre, plus particulièrement dans le cadre du Master « Logiciel libre » de l'URJC* [2]*. C'est un fervent défenseur de l'ouverture des ressources éducatives, du libre accès aux publications scientifiques et de l'ouverture des données scientifiques.*

Dans son célèbre essai *La Cathédrale et le Bazar* [1], Eric S. Raymond souligne l'une des plus importantes leçons que chaque développeur doit apprendre : « Un bon logiciel commence toujours par un développeur qui gratte là où ça le démange ». Vous ne pouvez comprendre à quel point cette phrase est vraie avant d'avoir vous-même vécu la situation. En fait, la majorité des programmeurs de logiciels libres et *open source* (si ce n'est tous) est certainement passée par cette étape où il faut mettre les mains dans le cambouis sur un tout nouveau projet, ou en rejoindre un, désireux d'aider à le rendre

meilleur. Cependant, de nombreux développeurs et autres participants dans les communautés libres et *open source* (rédacteurs de documentation, traducteurs etc.) négligent généralement une autre importante leçon soulignée par Raymond plus loin dans son essai : « Quand un programme ne vous intéresse plus, votre dernier devoir à son égard est de le confier à un successeur compétent ». C'est le thème central que je veux traiter ici. Vous devez penser à l'avenir de votre projet, et aux nouveaux arrivants qui un jour prendront le relais et continueront de le faire avancer.

Le relais entre les générations

Tôt ou tard, de nombreux projets libres et *open source* devront faire face à un relais générationnel. Les anciens développeurs en charge de la maintenance du code et des améliorations finissent par quitter le projet et sa communauté pour des raisons diverses et variées. Il peut s'agir de problèmes personnels, d'un nouveau travail qui ne laisse pas assez de disponibilités, d'un nouveau projet qui démarre, ou du passage à un autre projet qui semble plus attirant… la liste peut être assez longue.

L'étude du relais générationnel (ou renouvellement des développeurs) dans les projets de logiciel libre et *open source* reste un domaine émergent qui nécessite davantage de recherches pour améliorer notre compréhension de ces situations. En dépit de cela, certains chercheurs ont déjà collecté des preuves objectives qui mettent en lumière certains processus. Pendant l'OSS 2006 (NdT : Conférence sur l'Open Source System [4]), mes collègues Jesús González-Barahona et Gregorio Robles présentèrent un travail intitulé « Le renouvellement des contributeurs dans les projets de logiciel libre ». Dans cette présentation, ils exposèrent une méthodologie pour identifier les développeurs les plus actifs – généralement connus comme les développeurs principaux – à différents moments, pendant toute la durée d'un projet donné.

Ils appliquèrent ensuite cette méthode à l'étude de 21 gros projets, en particulier *Gimp* [5], *Mozilla* [6] et *Evolution* [7]. En bref, ils ont découvert qu'on peut distinguer trois types de projets en fonction du taux de renouvellement des développeurs.

- Les projets avec des dieux du code : ces projets reposent en grande partie sur le travail de leurs fondateurs et le relais générationnel est très faible, voire nul. *Gimp* se classe dans cette catégorie.
- Les projets avec de multiples générations de codeurs : des projets comme Mozilla montrent clairement un modèle de renouvellement des développeurs, avec de nouveaux groupes actifs qui prennent en main la gestion du développement et de la maintenance du code des mains mêmes du noyau des anciens contributeurs.
- Les projets composites : *Evolution* appartient à une troisième catégorie de projets ; il connaît un certain taux de renouvellement, mais celui-ci n'est toutefois pas aussi marqué que pour les projets de la catégorie précédente, parce qu'atténué par la rétention de certains des principaux contributeurs au cours de l'histoire du projet.

Cette classification nous amène à une question évidente : quel est le modèle le plus fréquemment rencontré dans les projets de logiciels libre et *open source* ? Pour tout dire, les résultats de l'analyse menée sur l'échantillon de 21 projets lors de ces travaux établissent clairement cette conclusion : ce sont les projets à multiples générations, ainsi que les projets composites qui sont les plus communément rencontrés dans l'écosystème des projets libres et *open source*. Seuls *Gnumeric* et *Mono* ont montré un modèle distinct avec une forte rétention d'anciens développeurs, ceci indiquant que les personnes contribuant à ces projets auraient de plus fortes raisons de continuer leurs travaux sur le long terme.

Cependant ce n’est pas le cas le plus courant. Au contraire, cette étude donne plus de légitimité au conseil suivant : nous devons préparer, à plus ou moins long terme, le transfert de notre rôle et de nos connaissances au sein du projet vers les futurs contributeurs qui rejoignent notre communauté.

Le fossé de connaissances

Toute personne faisant face à un changement significatif dans sa vie doit s’adapter à de nouvelles conditions. Par exemple, quand vous quittez votre emploi pour un autre, vous vous préparez à une période pendant laquelle il vous faudra vous intégrer dans un autre groupe de travail, dans un nouveau lieu. Heureusement, au bout d’un moment vous aurez pris vos marques dans ce nouvel emploi. Mais parfois vous aurez gardé de bons amis de votre ancien boulot, et vous vous reverrez après avoir bougé. Parfois alors, en discutant avec des anciens collègues, vous pourrez savoir ce qu’il est advenu avec la personne recrutée pour vous remplacer. Cela ne se produit que rarement dans les projets *open source*.

Le revers du relais générationnel dans un projet libre peut apparaitre sous une forme très concrète, à savoir un fossé de connaissances. Quand un ancien développeur quitte le projet, et particulièrement s’il avait une expérience approfondie dans cette communauté, il laisse derrière lui ses connaissances aussi bien concrètes qu’abstraites, qui ne sont pas forcément transmises aux nouveaux venus.

Un exemple évident, est le code source. Comme dans toute production intellectuelle bien faite – du moins, c’est ce à quoi on pourrait s’attendre, non ? – les développeurs laissent une marque personnelle chaque fois qu’ils écrivent du nouveau code. Parfois, vous avez l’impression d’avoir une dette éternelle envers le programmeur qui a écrit ce code élégant et soigné qui parle de lui-même et qui est facile à maintenir. D’autres fois, la situation est inverse et vous bataillez pour

comprendre un code très obscur sans un seul commentaire ni indice pour vous aider.

C'est ce que l'on a essayé de mesurer en 2009, dans une recherche présentée à l'HICSS 2009 (NdT : Hawaii International Conference on System Sciences [8]). Le titre en est « Utiliser l'archéologie logicielle pour mesurer les pertes de connaissances provoquées par le départ d'un développeur ». Au cas où vous vous poseriez la question, cela n'a rien à voir avec des histoires de fouet, de trésors, de temples, et autres aventures palpitantes. Ce qui a été mesuré, entre autres choses, c'est le pourcentage de code orphelin laissé par les développeurs ayant quitté des projets libre et/ou *open source*, et qu'aucun des développeurs actuels n'a encore repris. Pour cette étude, nous avons choisi quatre projets (*Evolution, GIMP, Evince* et *Nautilus*) pour tester la méthode de recherche. Et nous sommes arrivés à des résultats assez intéressants.

Evolution présentait une tendance plutôt inquiétante car le taux de code orphelin augmentait au cours du temps. En 2006, près de 80 % de l'ensemble des lignes de code avaient été abandonnées par les précédents développeurs et étaient restées intouchées par le reste de l'équipe. À l'opposé, *Gimp* affichait un modèle tout à fait différent, avec une volonté claire et soutenue par l'équipe de développement de réduire le nombre de lignes orphelines. Souvenons-nous au passage que *Gimp* avait déjà été qualifié de projet des dieux du code et bénéficiait donc d'une équipe de développement bien plus stable pour surmonter cette tâche harassante.

Cela signifie-t-il que les développeurs de *Gimp* avaient une bien meilleure expérience que ceux d'*Evolution* ? Honnêtement, on n'en sait rien. Néanmoins, on peut prévoir un risque clair : plus le taux de code orphelin est élevé, plus l'effort à fournir pour maintenir le projet est important. Que ce soit pour corriger un bogue, développer une nouvelle fonctionnalité ou en étendre une préexistante, il faut faire face à du code que l'on n'a jamais vu auparavant. Bien sûr les programmeurs

d’exception existent, mais peu importe à quel point l’on est merveilleux, les développeurs de *Gimp* ont un avantage certain ici, puisqu’ils ont quelqu’un dans l’équipe qui a une connaissance précise de la majorité du code à maintenir. De plus, ils travaillent à réduire la portion de code inconnu au cours du temps.

C’est comme à la maison

Ce qui est intéressant, c’est que certains projets parviennent à retenir les utilisateurs sur des périodes bien plus longues qu’on aurait pu s’y attendre. Là encore, nous pouvons trouver des preuves empiriques justifiant cette déclaration. Pendant l’OSS 2005 [9], Michlmayr, Robles et González-Barahona présentèrent des résultats pertinents concernant cet aspect. Ils étudièrent la persistance de la participation des responsables de logiciels sur Debian en calculant ce qu’on appelle la demi-vie. C’est le temps nécessaire à une population donnée de développeurs principaux pour perdre la moitié de sa taille initiale. Le résultat fut que la demi-vie estimée des responsables Debian était approximativement de 7 ans et demi. En d’autres termes, l’étude ayant été menée sur une période de six ans et demi (entre juillet 1998 et décembre 2004), donc depuis Debian 2.0 jusqu’à Debian 3.1 (versions stables uniquement), plus de 50 % des responsables de Debian 2.0 contribuaient encore à Debian 3.1.

Debian a créé une sorte de procédure très formelle pour accepter de nouveaux codeurs logiciels (aussi connus sous le nom de développeurs Debian) qui inclut l’acceptation du Contrat Social Debian et la démonstration d’une bonne connaissance de la Politique Debian. Résultat, on peut s’attendre à avoir des contributeurs très engagés. C’est en effet le cas, puisque les auteurs de l’étude ont constaté que les paquets délaissés par les anciens développeurs étaient généralement repris par d’autres développeurs de la communauté. C’est seulement dans le cas où le paquet n’était

plus utile qu'il a été abandonné. Je pense que nous pouvons tirer quelques conclusions de ces travaux de recherche :

1. Passez du temps à développer les principales lignes directrices de votre projet. Cela peut commencer par un seul et court document, qui détaille simplement des recommandations et des bonnes pratiques. Cela peut évoluer à mesure que le projet grandit, et permettre aux nouveaux arrivants tant de saisir rapidement les valeurs principales de votre équipe, qu'à comprendre les traits principaux de votre méthodologie.
2. Forcez-vous à suivre des standards de codage, des bonnes pratiques et un style élégant. Documentez votre code. Insérez des commentaires pour décrire les sections qui seraient particulièrement difficiles à comprendre. Ne pensez pas que c'est du temps perdu. En fait, vous faites preuve de pragmatisme en investissant du temps dans l'avenir de votre projet.
3. Dans la mesure du possible, lorsque le moment vient pour vous de quitter le projet, essayez d'avertir les autres de cette décision longtemps à l'avance. Assurez-vous qu'ils comprennent quelles parties essentielles du code nécessiteront un nouveau développeur pour le maintenir. Idéalement, si vous formez une communauté, préparez au moins une procédure simple afin d'automatiser la transition, et assurez-vous de n'oublier aucun point important avant que la personne ne quitte le projet (particulièrement si celle-ci est un développeur clé).
4. Gardez un œil sur la quantité de code orphelin. Si celle-ci augmente trop rapidement, ou si elle atteint une trop grande proportion de votre projet, cela indique clairement que vous allez avoir des problèmes dans peu de temps, en particulier si le nombre de rapports de bogues augmente ou si vous envisagez une nouvelle implémentation de votre code avec de fortes modifications.
5. Assurez-vous de toujours laisser assez d'astuces et de

commentaires pour qu'à l'avenir un nouvel arrivant puisse s'approprier votre travail.

J'aurais voulu savoir que vous arriviez (avant de partir)

Je reconnais que ce n'est pas très facile de penser à ses successeurs lorsque vous programmez. La plupart du temps, vous ne vous rendez tout simplement pas compte que votre code pourrait à la fin être repris par un autre projet, réutilisé par d'autres personnes ou que vous pourriez éventuellement être remplacé par un autre, qui continuera votre travail après vous. Cependant, le plus remarquable atout des logiciels libres et *open source* est précisément celui-là : le code sera réutilisé, adapté, intégré ou exporté par quelqu'un d'autre. La maintenance est une composante essentielle de l'ingénierie logicielle. Mais cela devient primordial dans le cas des logiciels libres et *open source*. Ce n'est pas seulement une question de code source. Cela concerne aussi les relations humaines et la netiquette. C'est quelque chose qui va au-delà du bon goût. *Quod severis metes* (« On récolte ce que l'on a semé »). Souvenez-vous en. La prochaine fois, vous pourriez être le nouveau venu qui viendra combler le vide de connaissance laissé par un ancien développeur.

[1] http://www.urjc.es

[2] http://www.urjc.es/estudios/masteres_universitarios/ingenieria/software_libre/index.htm

[3] http://www.linux-france.org/article/these/cathedrale-bazar/cathedrale-bazar.html

[4] http://oss2006.org

[5] logiciel de création graphique, http://www.gimp.org

[6] https://www.mozilla.org/fr/firefox/fx/

[7] logiciel de messagerie, http://projects.gnome.org/evolution

[8] archives de la conférence, http://www.informatik.uni-trier.de/~ley/db/conf/hicss/hicss2009.html

[9] http://oss2005.case.unibz.it

Traduction Framalang : ga3lig, Coco, Aa, Lignusta, goofy, jcr83, peupleLa (relectures), Sylvain, CoudCoud, lamessen + Julius22

Préparez-vous pour le futur : l'évolution des équipes dans le logiciel libre et *open source*

par Felipe Ortega

Felipe Ortega est chercheur et chef de projet à Libresoft, un groupe de recherche de l'Université Rey Juan Carlos [1] *en Espagne. Il développe de nouvelles méthodologies pour analyser les communautés collaboratives ouvertes (comme les projets de logiciels libres, Wikipédia et les réseaux sociaux). Il a mené des recherches approfondies sur le projet Wikipédia et sa communauté de contributeurs. Felipe participe activement à la recherche, la promotion et l'éducation/formation sur le logiciel libre, plus particulièrement dans le cadre du Master « Logiciel libre » de l'URJC* [2]*. C'est un fervent défenseur de l'ouverture des ressources éducatives, du libre accès aux publications scientifiques et de l'ouverture des données scientifiques.*

Dans son célèbre essai *La Cathédrale et le Bazar* [1], Eric S. Raymond souligne l'une des plus importantes leçons que chaque développeur doit apprendre : « Un bon logiciel commence toujours par un développeur qui gratte là où ça le démange ». Vous ne pouvez comprendre à quel point cette phrase est vraie avant d'avoir vous-même vécu la situation. En fait, la majorité des programmeurs de logiciels libres et *open source* (si ce n'est tous) est certainement passée par cette étape où il faut mettre les mains dans le cambouis sur un tout nouveau projet, ou en rejoindre un, désireux d'aider à le rendre meilleur. Cependant, de nombreux développeurs et autres participants dans les communautés libres et *open source* (rédacteurs de documentation, traducteurs etc.) négligent généralement une autre importante leçon soulignée par Raymond plus loin dans son essai : « Quand un programme ne vous intéresse plus, votre dernier devoir à son égard est de le confier à un successeur compétent ». C'est le thème central que je veux traiter ici. Vous devez penser à l'avenir de votre projet, et aux nouveaux arrivants qui un jour prendront le relais et continueront de le faire avancer.

Le relais entre les générations

Tôt ou tard, de nombreux projets libres et *open source* devront faire face à un relais générationnel. Les anciens développeurs en charge de la maintenance du code et des améliorations finissent par quitter le projet et sa communauté pour des raisons diverses et variées. Il peut s'agir de problèmes personnels, d'un nouveau travail qui ne laisse pas assez de disponibilités, d'un nouveau projet qui démarre, ou du passage à un autre projet qui semble plus attirant… la liste peut être assez longue.

L'étude du relais générationnel (ou renouvellement des développeurs) dans les projets de logiciel libre et *open source* reste un domaine émergent qui nécessite davantage de recherches pour améliorer notre compréhension de ces situations. En dépit de cela, certains chercheurs ont déjà collecté des preuves objectives qui mettent en lumière certains processus. Pendant l'OSS 2006 (NdT : Conférence sur l'Open Source System [4]), mes collègues Jesús González-Barahona et Gregorio Robles présentèrent un travail intitulé « Le renouvellement des contributeurs dans les projets de logiciel libre ». Dans cette présentation, ils exposèrent une méthodologie pour identifier les développeurs les plus actifs – généralement connus comme les développeurs principaux – à différents moments, pendant toute la durée d'un projet donné. Ils appliquèrent ensuite cette méthode à l'étude de 21 gros projets, en particulier *Gimp* [5], *Mozilla* [6] et *Evolution* [7]. En bref, ils ont découvert qu'on peut distinguer trois types de projets en fonction du taux de renouvellement des développeurs.

- Les projets avec des dieux du code : ces projets reposent en grande partie sur le travail de leurs fondateurs et le relais générationnel est très faible, voire nul. *Gimp* se classe dans cette catégorie.
- Les projets avec de multiples générations de codeurs :

des projets comme Mozilla montrent clairement un modèle de renouvellement des développeurs, avec de nouveaux groupes actifs qui prennent en main la gestion du développement et de la maintenance du code des mains mêmes du noyau des anciens contributeurs.

- Les projets composites : *Evolution* appartient à une troisième catégorie de projets ; il connaît un certain taux de renouvellement, mais celui-ci n'est toutefois pas aussi marqué que pour les projets de la catégorie précédente, parce qu'atténué par la rétention de certains des principaux contributeurs au cours de l'histoire du projet.

Cette classification nous amène à une question évidente : quel est le modèle le plus fréquemment rencontré dans les projets de logiciels libre et *open source* ? Pour tout dire, les résultats de l'analyse menée sur l'échantillon de 21 projets lors de ces travaux établissent clairement cette conclusion : ce sont les projets à multiples générations, ainsi que les projets composites qui sont les plus communément rencontrés dans l'écosystème des projets libres et *open source*. Seuls *Gnumeric* et *Mono* ont montré un modèle distinct avec une forte rétention d'anciens développeurs, ceci indiquant que les personnes contribuant à ces projets auraient de plus fortes raisons de continuer leurs travaux sur le long terme.

Cependant ce n'est pas le cas le plus courant. Au contraire, cette étude donne plus de légitimité au conseil suivant : nous devons préparer, à plus ou moins long terme, le transfert de notre rôle et de nos connaissances au sein du projet vers les futurs contributeurs qui rejoignent notre communauté.

Le fossé de connaissances

Toute personne faisant face à un changement significatif dans sa vie doit s'adapter à de nouvelles conditions. Par exemple, quand vous quittez votre emploi pour un autre, vous vous

préparez à une période pendant laquelle il vous faudra vous intégrer dans un autre groupe de travail, dans un nouveau lieu. Heureusement, au bout d'un moment vous aurez pris vos marques dans ce nouvel emploi. Mais parfois vous aurez gardé de bons amis de votre ancien boulot, et vous vous reverrez après avoir bougé. Parfois alors, en discutant avec des anciens collègues, vous pourrez savoir ce qu'il est advenu avec la personne recrutée pour vous remplacer. Cela ne se produit que rarement dans les projets *open source*.

Le revers du relais générationnel dans un projet libre peut apparaitre sous une forme très concrète, à savoir un fossé de connaissances. Quand un ancien développeur quitte le projet, et particulièrement s'il avait une expérience approfondie dans cette communauté, il laisse derrière lui ses connaissances aussi bien concrètes qu'abstraites, qui ne sont pas forcément transmises aux nouveaux venus.

Un exemple évident, est le code source. Comme dans toute production intellectuelle bien faite – du moins, c'est ce à quoi on pourrait s'attendre, non ? – les développeurs laissent une marque personnelle chaque fois qu'ils écrivent du nouveau code. Parfois, vous avez l'impression d'avoir une dette éternelle envers le programmeur qui a écrit ce code élégant et soigné qui parle de lui-même et qui est facile à maintenir. D'autres fois, la situation est inverse et vous bataillez pour comprendre un code très obscur sans un seul commentaire ni indice pour vous aider.

C'est ce que l'on a essayé de mesurer en 2009, dans une recherche présentée à l'HICSS 2009 (NdT : Hawaii International Conference on System Sciences [8]). Le titre en est « Utiliser l'archéologie logicielle pour mesurer les pertes de connaissances provoquées par le départ d'un développeur ». Au cas où vous vous poseriez la question, cela n'a rien à voir avec des histoires de fouet, de trésors, de temples, et autres aventures palpitantes. Ce qui a été mesuré, entre autres choses, c'est le pourcentage de code orphelin laissé par les

développeurs ayant quitté des projets libre et/ou *open source*, et qu'aucun des développeurs actuels n'a encore repris. Pour cette étude, nous avons choisi quatre projets (*Evolution, GIMP, Evince* et *Nautilus*) pour tester la méthode de recherche. Et nous sommes arrivés à des résultats assez intéressants.

Evolution présentait une tendance plutôt inquiétante car le taux de code orphelin augmentait au cours du temps. En 2006, près de 80 % de l'ensemble des lignes de code avaient été abandonnées par les précédents développeurs et étaient restées intouchées par le reste de l'équipe. À l'opposé, *Gimp* affichait un modèle tout à fait différent, avec une volonté claire et soutenue par l'équipe de développement de réduire le nombre de lignes orphelines. Souvenons-nous au passage que *Gimp* avait déjà été qualifié de projet des dieux du code et bénéficiait donc d'une équipe de développement bien plus stable pour surmonter cette tâche harassante.

Cela signifie-t-il que les développeurs de *Gimp* avaient une bien meilleure expérience que ceux d'*Evolution* ? Honnêtement, on n'en sait rien. Néanmoins, on peut prévoir un risque clair : plus le taux de code orphelin est élevé, plus l'effort à fournir pour maintenir le projet est important. Que ce soit pour corriger un bogue, développer une nouvelle fonctionnalité ou en étendre une préexistante, il faut faire face à du code que l'on n'a jamais vu auparavant. Bien sûr les programmeurs d'exception existent, mais peu importe à quel point l'on est merveilleux, les développeurs de *Gimp* ont un avantage certain ici, puisqu'ils ont quelqu'un dans l'équipe qui a une connaissance précise de la majorité du code à maintenir. De plus, ils travaillent à réduire la portion de code inconnu au cours du temps.

C'est comme à la maison

Ce qui est intéressant, c'est que certains projets parviennent à retenir les utilisateurs sur des périodes bien plus longues

qu’on aurait pu s’y attendre. Là encore, nous pouvons trouver des preuves empiriques justifiant cette déclaration. Pendant l’OSS 2005 [9], Michlmayr, Robles et González-Barahona présentèrent des résultats pertinents concernant cet aspect. Ils étudièrent la persistance de la participation des responsables de logiciels sur Debian en calculant ce qu’on appelle la demi-vie. C’est le temps nécessaire à une population donnée de développeurs principaux pour perdre la moitié de sa taille initiale. Le résultat fut que la demi-vie estimée des responsables Debian était approximativement de 7 ans et demi. En d’autres termes, l’étude ayant été menée sur une période de six ans et demi (entre juillet 1998 et décembre 2004), donc depuis Debian 2.0 jusqu’à Debian 3.1 (versions stables uniquement), plus de 50 % des responsables de Debian 2.0 contribuaient encore à Debian 3.1.

Debian a créé une sorte de procédure très formelle pour accepter de nouveaux codeurs logiciels (aussi connus sous le nom de développeurs Debian) qui inclut l’acceptation du Contrat Social Debian et la démonstration d’une bonne connaissance de la Politique Debian. Résultat, on peut s’attendre à avoir des contributeurs très engagés. C’est en effet le cas, puisque les auteurs de l’étude ont constaté que les paquets délaissés par les anciens développeurs étaient généralement repris par d’autres développeurs de la communauté. C’est seulement dans le cas où le paquet n’était plus utile qu’il a été abandonné. Je pense que nous pouvons tirer quelques conclusions de ces travaux de recherche :

1. Passez du temps à développer les principales lignes directrices de votre projet. Cela peut commencer par un seul et court document, qui détaille simplement des recommandations et des bonnes pratiques. Cela peut évoluer à mesure que le projet grandit, et permettre aux nouveaux arrivants tant de saisir rapidement les valeurs principales de votre équipe, qu’à comprendre les traits principaux de votre méthodologie.

2. Forcez-vous à suivre des standards de codage, des bonnes pratiques et un style élégant. Documentez votre code. Insérez des commentaires pour décrire les sections qui seraient particulièrement difficiles à comprendre. Ne pensez pas que c'est du temps perdu. En fait, vous faites preuve de pragmatisme en investissant du temps dans l'avenir de votre projet.
3. Dans la mesure du possible, lorsque le moment vient pour vous de quitter le projet, essayez d'avertir les autres de cette décision longtemps à l'avance. Assurez-vous qu'ils comprennent quelles parties essentielles du code nécessiteront un nouveau développeur pour le maintenir. Idéalement, si vous formez une communauté, préparez au moins une procédure simple afin d'automatiser la transition, et assurez-vous de n'oublier aucun point important avant que la personne ne quitte le projet (particulièrement si celle-ci est un développeur clé).
4. Gardez un œil sur la quantité de code orphelin. Si celle-ci augmente trop rapidement, ou si elle atteint une trop grande proportion de votre projet, cela indique clairement que vous allez avoir des problèmes dans peu de temps, en particulier si le nombre de rapports de bogues augmente ou si vous envisagez une nouvelle implémentation de votre code avec de fortes modifications.
5. Assurez-vous de toujours laisser assez d'astuces et de commentaires pour qu'à l'avenir un nouvel arrivant puisse s'approprier votre travail.

J'aurais voulu savoir que vous arriviez (avant de partir)

Je reconnais que ce n'est pas très facile de penser à ses successeurs lorsque vous programmez. La plupart du temps, vous ne vous rendez tout simplement pas compte que votre code pourrait à la fin être repris par un autre projet, réutilisé

par d'autres personnes ou que vous pourriez éventuellement être remplacé par un autre, qui continuera votre travail après vous. Cependant, le plus remarquable atout des logiciels libres et *open source* est précisément celui-là : le code sera réutilisé, adapté, intégré ou exporté par quelqu'un d'autre. La maintenance est une composante essentielle de l'ingénierie logicielle. Mais cela devient primordial dans le cas des logiciels libres et *open source*. Ce n'est pas seulement une question de code source. Cela concerne aussi les relations humaines et la netiquette. C'est quelque chose qui va au-delà du bon goût. *Quod severis metes* (« On récolte ce que l'on a semé »). Souvenez-vous en. La prochaine fois, vous pourriez être le nouveau venu qui viendra combler le vide de connaissance laissé par un ancien développeur.

[1] http://www.urjc.es

[2] http://www.urjc.es/estudios/masteres_universitarios/ingenieria/software_libre/index.htm

[3] http://www.linux-france.org/article/these/cathedrale-bazar/cathedrale-bazar.html

[4] http://oss2006.org

[5] logiciel de création graphique, http://www.gimp.org

[6] https://www.mozilla.org/fr/firefox/fx/

[7] logiciel de messagerie, http://projects.gnome.org/evolution

[8] archives de la conférence, http://www.informatik.uni-trier.de/~ley/db/conf/hicss/hicss2009.html

[9] http://oss2005.case.unibz.it

Framasoft et le Père Gnuël vous souhaitent de joyeuses fêtes !

Avec ce sourire, nous en profitons pour vous remercier chaleureusement de votre soutien cette année, qu'il s'agisse d'un don[1], d'une participation ou tout simplement en diffusant *la bonne parole du Libre* autour de vous.

Rendez-vous en 2013, parce qu'il reste encore plein de *sales gosses* à convertir

Crédit : Simon Gee Giraudot (Creative Commons By-Sa)

Notes

[1] Pour rappel : C'est le dernier moment pour faire un don 2012 défiscalisable à 66% (si vous êtes soumis à l'impôt sur le revenu).

Les projets open source s'épanouissent à l'air libre (Libres conseils 3/42)

3. Hors du labo, au grand air

La croissance des communautés open source autour des projets universitaires

par Markus Kroëtzsch

Markus Kroëtzsch est post-doctorant au Département des Sciences Informatiques de l'Université d'Oxford. Il a soutenu son doctorat en 2010 à l'Institut d'Informatique Appliquée et Méthodes Formelles de description du Karlsruhe Institute of Technology. Ses recherches portent sur le traitement automatique de l'information, depuis les fondements de la représentation de la connaissance formelle jusqu'à leurs domaines d'application, tel le Web Sémantique. Il est le développeur principal de la plate-forme Semantic MediaWiki, application de Web sémantique, co-éditeur des spécifications

W3C OWL2, administrateur principal du portail communautaire semanticweb.org, et co-auteur de l'ouvrage Foundations of Semantic Web Technologies.

Au sein des universités, les chercheurs développent de grandes quantités de logiciels, que ce soit pour valider une hypothèse, pour illustrer une nouvelle approche, ou tout simplement comme outil en appui à une étude. Dans la plupart des cas, un petit prototype dédié fait le travail, et il est déployé rapidement tandis que l'enjeu de la recherche évolue. Cependant, de temps à autres, une nouvelle approche ou une technologie émergente a le potentiel de changer complétement la manière de résoudre un problème. Ce faisant, cela génère de la réputation professionnelle, du succès commercial, et la gratification personnelle d'amener une nouvelle idée à son plein potentiel. Le chercheur qui a fait une découverte de ce genre est alors tenté d'aller au-delà du prototype vers un produit qui sera réellement utilisé. Il est alors confronté à une toute nouvelle série de problèmes pratiques.

La peur de l'utilisateur

Dans l'un de ses célèbres essais sur l'ingénierie logicielle, Frederick P. Brooks, Jr. permet de se faire une bonne idée des efforts liés à la maintenance d'un vrai logiciel et nous met en garde contre l'utilisateur :

« Le coût total de la maintenance d'un programme largement utilisé est habituellement de 40% ou plus de son coût de développement. De façon surprenante, ce coût est fortement influencé par le nombre d'utilisateurs. Plus il y a d'utilisateurs, plus il y a de bugs » [1]

Bien que les chiffres soient probablement différents dans le contexte actuel, la remarque reste fondamentalement vraie. Elle pourrait même avoir été confirmée par l'usage généralisé de la communication instantanée. Pire encore : davantage d'utilisateurs ne produit pas seulement davantage de bugs ; en

général, ils expriment aussi plus de demandes. Qu'il s'agisse d'une véritable erreur, d'une demande de fonctionnalité, ou tout simplement d'une mauvaise compréhension du fonctionnement du logiciel, les demandes de l'utilisateur lambda ne ressemblent en rien à un rapport de bug précis et technique. Et chaque demande requiert l'attention des développeurs et occupe un temps précieux qui n'est plus disponible pour écrire du code.

Avec son esprit d'analyse, le chercheur anticipe sur ce problème. Dans sa lutte naturelle pour éviter un avenir sombre dans le service client, il peut carrément développer de la peur envers l'utilisateur. Dans le pire des cas, cela peut le mener à prendre des décisions qui vont à l'encontre du projet dans son ensemble ; sous des formes plus légères, cela peut tout de même mener le chercheur à cacher des produits logiciels brillants à ses utilisateurs potentiels. Plus d'une fois, j'ai entendu des chercheurs dire : « Nous n'avons pas besoin de plus de visibilité : nous recevons déjà suffisamment d'emails ! ». Il est vrai que parfois la communication pour un outil logiciel nécessite un effort supérieur à celui que peut fournir un chercheur sans laisser tomber son emploi principal.

Pourtant, cette issue tragique aurait bien souvent pu être évitée. Brooks pouvait difficilement l'anticiper. Quand il a écrit ses essais, le fait est que les utilisateurs étaient des clients et que la maintenance logicielle faisait partie du produit qu'ils achetaient. Il fallait trouver un équilibre entre l'effort de développement, la demande du marché, et le prix. C'est toujours le cas pour de nombreux produits logiciels commerciaux de nos jours, mais ça a peu de choses à voir avec la réalité du développement à petite échelle de l'*open source*. Les utilisateurs habituels de l'*open source* ne paient pas pour le service qu'ils reçoivent. Leur attitude n'est donc pas celle d'un client exigeant, mais bien plus souvent celle d'un partisan enthousiaste et reconnaissant. Transformer cet enthousiasme en un soutien plus que nécessaire

n’est pas négligeable pour réussir dans l’art de la maintenance d’un logiciel open source : l’intérêt croissant de l’utilisateur doit aller de pair avec une contribution croissante.

Reconnaitre que les utilisateurs de logiciels *open source* ne sont pas que « des consommateurs qui ne paient pas » est une notion importante. Mais cela ne doit pas mener à surestimer leur potentiel. Le contre-pied optimiste de la peur irrationnelle de l’utilisateur est la croyance que des communautés actives croissent naturellement avec pour seule base la licence choisie pour publier le code. Cette grave erreur de jugement est bizarrement toujours aussi commune, et a scellé le destin de bien des tentatives de création de communautés ouvertes.

Semer et récolter

Le pluriel d’ « utilisateur » n’est pas « communauté ». Si l’un peut s’accroître en nombre, l’autre ne grandit pas d’elle-même, ou alors elle grandit sans direction et surtout sans fournir le soutien espéré au projet. La mission du responsable de projet qui cherche à profiter de l’énergie brute des utilisateurs ressemble à celle d’un jardinier qui doit préparer un terrain fertile, planter et arroser les semis, et peut-être élaguer les pousses non désirées avant de pouvoir récolter les fruits. Par rapport aux récompenses, l’investissement global est minime, mais il est essentiel de faire les bonnes choses, au bon moment.

Préparer le support technologique

Créer une communauté commence avant même que le premier utilisateur n’apparaisse. D’emblée, le choix du langage de programmation va déterminer le nombre de personnes qui pourront déployer et déboguer notre code. Objective Caml est sans doute un beau langage, mais si l’on utilise plutôt Java,

la quantité d'utilisateurs et de contributeurs potentiels augmentera de plusieurs ordres de grandeur. Les développeurs doivent donc faire des compromis puisque la technologie la plus répandue est rarement la plus performante ou la plus élégante. Cela peut être une démarche particulièrement difficile pour des chercheurs qui privilégient souvent la supériorité formelle du langage. Quand je travaillais sur Semantic MediaWiki, on m'a souvent demandé pourquoi nous utilisions PHP alors que Java côté serveur serait tellement plus propre et performant. Comparons la taille de la communauté de Semantic MediaWiki et les efforts que demanderait le même développement basé sur du Java : voilà peut-être un début de réponse. Cet exemple illustre aussi que l'audience ciblée détermine le meilleur choix de la technologie de base. Le développeur lui-même devrait avoir le recul nécessaire pour prendre la décision la plus opportune.

Préparer minutieusement le terrain

Dans le même ordre d'idées, il faut créer un code lisible et bien documenté dès le début. Dans un environnement universitaire, certains projets logiciels rassemblent de nombreux contributeurs temporaires. Les changements dans les plannings et les projets des étudiants peuvent nuire à la qualité du code. Je me souviens d'un petit projet de logiciel à l'Université technique de Dresde qui avait été très bien maintenu par un assistant étudiant. Après son départ, on a constaté que le code était minutieusement documenté… en turc. Un chercheur ne peut être programmeur qu'à temps partiel, une discipline particulière est donc nécessaire pour mettre en œuvre le travail supplémentaire indispensable à l'élaboration d'un code accessible. En retour il y aura de bien meilleures chances par la suite d'avoir de bons rapports de bogues, des patchs utiles ou même des développeurs extérieurs.

Semer les graines des communautés

Les développeurs *open source* inexpérimentés considèrent souvent comme un grand moment la publication ouverte de leur code. En réalité, personne d'autre qu'eux ne la remarquera. Pour attirer aussi bien des utilisateurs que des contributeurs, il faut faire passer le mot. La communication publique d'un vrai projet devrait au moins inclure des annonces à chaque nouvelle version. Les listes de diffusion sont probablement le meilleur canal pour cela. Il faut un certain talent social pour trouver le juste équilibre entre le spam indésirable et la litote timide. Si un projet est motivé par la conviction sincère qu'il aidera les utilisateurs à résoudre de vrais problèmes, ce devrait être facile de lui faire une publicité convenable. Les utilisateurs feront vite la différence entre publicité éhontée et information utile. Bien évidemment, les annonces actives devront attendre que le projet soit finalisé. Cela ne concerne pas seulement le code, mais aussi la page d'accueil et la documentation d'utilisation basique.

Au cours de sa vie, le projet devrait être mentionné dans tous les endroits adéquats, y compris des sites web – à commencer par votre page d'accueil ! – des conférences, des papiers scientifiques, des discussions en ligne. On ne dira jamais assez le pouvoir du simple lien qui conduira un futur contributeur important sur le site du projet dès sa première visite. Les chercheurs ne doivent pas non plus oublier de publier leur logiciel en dehors de leur communauté universitaire proche. Les autres chercheurs sont rarement la meilleure base pour une communauté active.

Fournir des espaces pour grandir

Banal mais souvent négligé, le devoir des personnes qui maintiennent le projet est de fournir des espaces de communication afin que la communauté puisse se développer. Si

un projet n’a pas de liste de discussion dédiée, alors toutes les demandes d’aide seront envoyées en message privé à la maintenance. S’il n’y a pas de bugtracker (NdT : logiciel de suivi de problèmes), les rapports de bogues seront moins nombreux et moins utiles. Sans un wiki éditable par tout un chacun pour la documentation utilisateur, le développeur est condamné à étendre et à réécrire la documentation en permanence. Si la version de développement du code source n’est pas accessible, alors les utilisateurs ne seront pas dans la capacité de tester la dernière version avant de se plaindre de problèmes. Si le dépôt de code est intrinsèquement fermé, il n’est alors pas possible d’accueillir des contributeurs externes. Toute cette infrastructure est disponible gratuitement par l’intermédiaire d’un certain nombre de fournisseurs de service. Toutes les formes d’interaction ne sont pas forcément désirées, par exemple, il y a des raisons de garder fermé le cercle des développeurs. Mais il serait inconscient d’espérer le soutien d’une communauté sans même préparer des espaces de base pour celle-ci.

Encourager et contrôler la croissance

Les développeurs inexpérimentés sont souvent préoccupés par le fait que l’ouverture de listes de diffusions, de forums et de wikis pour les utilisateurs nécessitera une maintenance supplémentaire. C’est rarement le cas, mais certaines activités de base sont bien entendu indispensables. Cela commence par la mise en œuvre rigoureuse des usages de communication publique. Les utilisateurs ont besoin d’apprendre à poser des questions publiquement, à consulter la documentation avant de poser des questions, et à rapporter les bogues à l’aide du bugtracker plutôt que par e-mail. J’ai tendance à rejeter toutes les demandes d’aide privées, ou à répondre via des listes publiques. Cela garantit au passage

que les solutions sont disponibles sur le web pour les futurs utilisateurs qui les chercheront. Dans tous les cas, les utilisateurs doivent être remerciés explicitement pour toutes les formes de contributions : il faut beaucoup d'enthousiasme et des gens bien intentionnés pour construire une communauté solide.

Quand on atteint un certain nombre d'utilisateurs, une aide mutuelle commence à se mettre en place entre eux. C'est toujours un moment magique pour un projet et c'est un signe évident qu'il est sur la bonne voie. Dans l'idéal, les responsables du projet devraient continuer d'apporter leur aide pour les questions délicates, mais à un moment donné certains utilisateurs vont faire preuve d'initiative dans les discussions. Il est important de les remercier (personnellement) et de les impliquer d'avantage dans le projet. À l'inverse, les évolutions malsaines doivent être stoppées dès que possible, en particulier les comportements agressifs qui peuvent être un véritable danger pour le développement de la communauté. De même, l'enthousiasme le mieux intentionné n'est pas toujours productif et il faut parfois savoir dire non – gentiment mais clairement – pour éviter les dérapages possibles.

Le futur est ouvert

Construire une communauté initiale autour d'un projet contribue fortement à transformer un prototype de recherche en un logiciel open source. Si ça porte ses fruits, il existe de nombreuses options pour le développer en fonction des buts fixés par le mainteneur du projet et la communauté. Voici quelques indications :

Persévérer dans l'expansion du projet et de sa communauté open source, augmenter le nombre de personnes ayant des droits de contributions « directs », en réduisant la dépendance à son origine universitaire. Par ce biais, vous impliquez plus

fortement la communauté – notamment au travers d'événements dédiés –;et vous pourrez établir un soutien organisationnel.

Créer une entreprise commerciale pour exploiter le projet, basée, par exemple, sur une double licence ou un *business model* de consultant. Des outils ayant fait leurs preuves et une communauté active sont des atouts majeurs dès le lancement d'une entreprise, et peuvent être bénéfiques dans chacune de vos stratégies d'entreprise sans abandonner le produit original open source.

Se retirer du projet. Il y a de nombreuses raisons pour qu'on ne puisse plus maintenir de lien avec le projet. Le fait d'avoir établi une communauté saine et ouverte maximise les chances pour que le projet continue de voler de ses propres ailes. Dans tous les cas, il est plus correct de faire une coupure nette que d'abandonner silencieusement le projet, en le tuant par une activité en chute libre au point qu'on ne trouve plus personne pour le maintenir.

Le profil de la communauté sera différent selon qu'on opte pour telle ou telle stratégie de développement. Mais dans tous les cas, le rôle du chercheur évolue en fonction des objectifs du projet. Le scientifique initial et le programmeur pourront prendre le rôle de gestionnaire ou de directeur technique. En ce sens, la différence principale entre un projet de logiciel open source d'importance et la recherche perpétuelle d'un prototype n'est pas tant la quantité de travail mais le type de travail nécessaire pour y arriver. Cela compte pour beaucoup dans sa réussite. Une fois qu'on l'a compris, il ne reste plus qu'à réaliser un super logiciel.

[1] Frederick P. Brooks, Jr.: The Mythical Man-Month. Essays on Software Engineering. Anniversary Edition. Addison-Wesley, 1995.

3. Hors du labo, au grand air

La croissance des communautés open source autour des projets universitaires

par Markus Kroëtzsch

Markus Kroëtzsch est post-doctorant au Département des Sciences Informatiques de l'Université d'Oxford. Il a soutenu son doctorat en 2010 à l'Institut d'Informatique Appliquée et Méthodes Formelles de description du Karlsruhe Institute of Technology. Ses recherches portent sur le traitement automatique de l'information, depuis les fondements de la représentation de la connaissance formelle jusqu'à leurs domaines d'application, tel le Web Sémantique. Il est le développeur principal de la plate-forme Semantic MediaWiki, application de Web sémantique, co-éditeur des spécifications W3C OWL2, administrateur principal du portail communautaire semanticweb.org, et co-auteur de l'ouvrage Foundations of Semantic Web Technologies.

Au sein des universités, les chercheurs développent de grandes quantités de logiciels, que ce soit pour valider une hypothèse, pour illustrer une nouvelle approche, ou tout simplement comme outil en appui à une étude. Dans la plupart des cas, un petit prototype dédié fait le travail, et il est déployé rapidement tandis que l'enjeu de la recherche évolue. Cependant, de temps à autres, une nouvelle approche ou une technologie émergente a le potentiel de changer complétement la manière de résoudre un problème. Ce faisant, cela génère de la réputation professionnelle, du succès commercial, et la gratification personnelle d'amener une nouvelle idée à son plein potentiel. Le chercheur qui a fait une découverte de ce genre est alors tenté d'aller au-delà du prototype vers un

produit qui sera réellement utilisé. Il est alors confronté à une toute nouvelle série de problèmes pratiques.

La peur de l'utilisateur

Dans l'un de ses célèbres essais sur l'ingénierie logicielle, Frederick P. Brooks, Jr. permet de se faire une bonne idée des efforts liés à la maintenance d'un vrai logiciel et nous met en garde contre l'utilisateur :

« Le coût total de la maintenance d'un programme largement utilisé est habituellement de 40% ou plus de son coût de développement. De façon surprenante, ce coût est fortement influencé par le nombre d'utilisateurs. Plus il y a d'utilisateurs, plus il y a de bugs » [1]

Bien que les chiffres soient probablement différents dans le contexte actuel, la remarque reste fondamentalement vraie. Elle pourrait même avoir été confirmée par l'usage généralisé de la communication instantanée. Pire encore : davantage d'utilisateurs ne produit pas seulement davantage de bugs ; en général, ils expriment aussi plus de demandes. Qu'il s'agisse d'une véritable erreur, d'une demande de fonctionnalité, ou tout simplement d'une mauvaise compréhension du fonctionnement du logiciel, les demandes de l'utilisateur lambda ne ressemblent en rien à un rapport de bug précis et technique. Et chaque demande requiert l'attention des développeurs et occupe un temps précieux qui n'est plus disponible pour écrire du code.

Avec son esprit d'analyse, le chercheur anticipe sur ce problème. Dans sa lutte naturelle pour éviter un avenir sombre dans le service client, il peut carrément développer de la peur envers l'utilisateur. Dans le pire des cas, cela peut le mener à prendre des décisions qui vont à l'encontre du projet dans son ensemble ; sous des formes plus légères, cela peut tout de même mener le chercheur à cacher des produits logiciels brillants à ses utilisateurs potentiels. Plus d'une

fois, j’ai entendu des chercheurs dire : « Nous n’avons pas besoin de plus de visibilité : nous recevons déjà suffisamment d’emails ! ». Il est vrai que parfois la communication pour un outil logiciel nécessite un effort supérieur à celui que peut fournir un chercheur sans laisser tomber son emploi principal.

Pourtant, cette issue tragique aurait bien souvent pu être évitée. Brooks pouvait difficilement l’anticiper. Quand il a écrit ses essais, le fait est que les utilisateurs étaient des clients et que la maintenance logicielle faisait partie du produit qu’ils achetaient. Il fallait trouver un équilibre entre l’effort de développement, la demande du marché, et le prix. C’est toujours le cas pour de nombreux produits logiciels commerciaux de nos jours, mais ça a peu de choses à voir avec la réalité du développement à petite échelle de l’*open source*. Les utilisateurs habituels de l’*open source* ne paient pas pour le service qu’ils reçoivent. Leur attitude n’est donc pas celle d’un client exigeant, mais bien plus souvent celle d’un partisan enthousiaste et reconnaissant. Transformer cet enthousiasme en un soutien plus que nécessaire n’est pas négligeable pour réussir dans l’art de la maintenance d’un logiciel open source : l’intérêt croissant de l’utilisateur doit aller de pair avec une contribution croissante.

Reconnaitre que les utilisateurs de logiciels *open source* ne sont pas que « des consommateurs qui ne paient pas » est une notion importante. Mais cela ne doit pas mener à surestimer leur potentiel. Le contre-pied optimiste de la peur irrationnelle de l’utilisateur est la croyance que des communautés actives croissent naturellement avec pour seule base la licence choisie pour publier le code. Cette grave erreur de jugement est bizarrement toujours aussi commune, et a scellé le destin de bien des tentatives de création de communautés ouvertes.

Semer et récolter

Le pluriel d’ « utilisateur » n’est pas « communauté ». Si l’un peut s’accroître en nombre, l’autre ne grandit pas d’elle-même, ou alors elle grandit sans direction et surtout sans fournir le soutien espéré au projet. La mission du responsable de projet qui cherche à profiter de l’énergie brute des utilisateurs ressemble à celle d’un jardinier qui doit préparer un terrain fertile, planter et arroser les semis, et peut-être élaguer les pousses non désirées avant de pouvoir récolter les fruits. Par rapport aux récompenses, l’investissement global est minime, mais il est essentiel de faire les bonnes choses, au bon moment.

Préparer le support technologique

Créer une communauté commence avant même que le premier utilisateur n’apparaisse. D’emblée, le choix du langage de programmation va déterminer le nombre de personnes qui pourront déployer et déboguer notre code. Objective Caml est sans doute un beau langage, mais si l’on utilise plutôt Java, la quantité d’utilisateurs et de contributeurs potentiels augmentera de plusieurs ordres de grandeur. Les développeurs doivent donc faire des compromis puisque la technologie la plus répandue est rarement la plus performante ou la plus élégante. Cela peut être une démarche particulièrement difficile pour des chercheurs qui privilégient souvent la supériorité formelle du langage. Quand je travaillais sur Semantic MediaWiki, on m’a souvent demandé pourquoi nous utilisions PHP alors que Java côté serveur serait tellement plus propre et performant. Comparons la taille de la communauté de Semantic MediaWiki et les efforts que demanderait le même développement basé sur du Java : voilà peut-être un début de réponse. Cet exemple illustre aussi que l’audience ciblée détermine le meilleur choix de la technologie de base. Le développeur lui-même devrait avoir le

recul nécessaire pour prendre la décision la plus opportune.

Préparer minutieusement le terrain

Dans le même ordre d'idées, il faut créer un code lisible et bien documenté dès le début. Dans un environnement universitaire, certains projets logiciels rassemblent de nombreux contributeurs temporaires. Les changements dans les plannings et les projets des étudiants peuvent nuire à la qualité du code. Je me souviens d'un petit projet de logiciel à l'Université technique de Dresde qui avait été très bien maintenu par un assistant étudiant. Après son départ, on a constaté que le code était minutieusement documenté… en turc. Un chercheur ne peut être programmeur qu'à temps partiel, une discipline particulière est donc nécessaire pour mettre en œuvre le travail supplémentaire indispensable à l'élaboration d'un code accessible. En retour il y aura de bien meilleures chances par la suite d'avoir de bons rapports de bogues, des patchs utiles ou même des développeurs extérieurs.

Semer les graines des communautés

Les développeurs *open source* inexpérimentés considèrent souvent comme un grand moment la publication ouverte de leur code. En réalité, personne d'autre qu'eux ne la remarquera. Pour attirer aussi bien des utilisateurs que des contributeurs, il faut faire passer le mot. La communication publique d'un vrai projet devrait au moins inclure des annonces à chaque nouvelle version. Les listes de diffusion sont probablement le meilleur canal pour cela. Il faut un certain talent social pour trouver le juste équilibre entre le spam indésirable et la litote timide. Si un projet est motivé par la conviction sincère qu'il aidera les utilisateurs à résoudre de vrais problèmes, ce devrait être facile de lui faire une publicité convenable. Les utilisateurs feront vite la différence entre publicité éhontée et information utile.

Bien évidemment, les annonces actives devront attendre que le projet soit finalisé. Cela ne concerne pas seulement le code, mais aussi la page d'accueil et la documentation d'utilisation basique.

Au cours de sa vie, le projet devrait être mentionné dans tous les endroits adéquats, y compris des sites web – à commencer par votre page d'accueil ! – des conférences, des papiers scientifiques, des discussions en ligne. On ne dira jamais assez le pouvoir du simple lien qui conduira un futur contributeur important sur le site du projet dès sa première visite. Les chercheurs ne doivent pas non plus oublier de publier leur logiciel en dehors de leur communauté universitaire proche. Les autres chercheurs sont rarement la meilleure base pour une communauté active.

Fournir des espaces pour grandir

Banal mais souvent négligé, le devoir des personnes qui maintiennent le projet est de fournir des espaces de communication afin que la communauté puisse se développer. Si un projet n'a pas de liste de discussion dédiée, alors toutes les demandes d'aide seront envoyées en message privé à la maintenance. S'il n'y a pas de bugtracker (NdT : logiciel de suivi de problèmes), les rapports de bogues seront moins nombreux et moins utiles. Sans un wiki éditable par tout un chacun pour la documentation utilisateur, le développeur est condamné à étendre et à réécrire la documentation en permanence. Si la version de développement du code source n'est pas accessible, alors les utilisateurs ne seront pas dans la capacité de tester la dernière version avant de se plaindre de problèmes. Si le dépôt de code est intrinsèquement fermé, il n'est alors pas possible d'accueillir des contributeurs externes. Toute cette infrastructure est disponible gratuitement par l'intermédiaire d'un certain nombre de fournisseurs de service. Toutes les formes d'interaction ne sont pas forcément désirées, par exemple, il

y a des raisons de garder fermé le cercle des développeurs. Mais il serait inconscient d'espérer le soutien d'une communauté sans même préparer des espaces de base pour celle-ci.

Encourager et contrôler la croissance

Les développeurs inexpérimentés sont souvent préoccupés par le fait que l'ouverture de listes de diffusions, de forums et de wikis pour les utilisateurs nécessitera une maintenance supplémentaire. C'est rarement le cas, mais certaines activités de base sont bien entendu indispensables. Cela commence par la mise en œuvre rigoureuse des usages de communication publique. Les utilisateurs ont besoin d'apprendre à poser des questions publiquement, à consulter la documentation avant de poser des questions, et à rapporter les bogues à l'aide du bugtracker plutôt que par e-mail. J'ai tendance à rejeter toutes les demandes d'aide privées, ou à répondre via des listes publiques. Cela garantit au passage que les solutions sont disponibles sur le web pour les futurs utilisateurs qui les chercheront. Dans tous les cas, les utilisateurs doivent être remerciés explicitement pour toutes les formes de contributions : il faut beaucoup d'enthousiasme et des gens bien intentionnés pour construire une communauté solide.

Quand on atteint un certain nombre d'utilisateurs, une aide mutuelle commence à se mettre en place entre eux. C'est toujours un moment magique pour un projet et c'est un signe évident qu'il est sur la bonne voie. Dans l'idéal, les responsables du projet devraient continuer d'apporter leur aide pour les questions délicates, mais à un moment donné certains utilisateurs vont faire preuve d'initiative dans les discussions. Il est important de les remercier (personnellement) et de les impliquer d'avantage dans le

projet. À l'inverse, les évolutions malsaines doivent être stoppées dès que possible, en particulier les comportements agressifs qui peuvent être un véritable danger pour le développement de la communauté. De même, l'enthousiasme le mieux intentionné n'est pas toujours productif et il faut parfois savoir dire non – gentiment mais clairement – pour éviter les dérapages possibles.

Le futur est ouvert

Construire une communauté initiale autour d'un projet contribue fortement à transformer un prototype de recherche en un logiciel open source. Si ça porte ses fruits, il existe de nombreuses options pour le développer en fonction des buts fixés par le mainteneur du projet et la communauté. Voici quelques indications :

Persévérer dans l'expansion du projet et de sa communauté open source, augmenter le nombre de personnes ayant des droits de contributions « directs », en réduisant la dépendance à son origine universitaire. Par ce biais, vous impliquez plus fortement la communauté – notamment au travers d'événements dédiés –;et vous pourrez établir un soutien organisationnel.

Créer une entreprise commerciale pour exploiter le projet, basée, par exemple, sur une double licence ou un *business model* de consultant. Des outils ayant fait leurs preuves et une communauté active sont des atouts majeurs dès le lancement d'une entreprise, et peuvent être bénéfiques dans chacune de vos stratégies d'entreprise sans abandonner le produit original open source.

Se retirer du projet. Il y a de nombreuses raisons pour qu'on ne puisse plus maintenir de lien avec le projet. Le fait d'avoir établi une communauté saine et ouverte maximise les chances pour que le projet continue de voler de ses propres ailes. Dans tous les cas, il est plus correct de faire une coupure nette que d'abandonner silencieusement le projet, en

le tuant par une activité en chute libre au point qu'on ne trouve plus personne pour le maintenir.

Le profil de la communauté sera différent selon qu'on opte pour telle ou telle stratégie de développement. Mais dans tous les cas, le rôle du chercheur évolue en fonction des objectifs du projet. Le scientifique initial et le programmeur pourront prendre le rôle de gestionnaire ou de directeur technique. En ce sens, la différence principale entre un projet de logiciel open source d'importance et la recherche perpétuelle d'un prototype n'est pas tant la quantité de travail mais le type de travail nécessaire pour y arriver. Cela compte pour beaucoup dans sa réussite. Une fois qu'on l'a compris, il ne reste plus qu'à réaliser un super logiciel.

[1] Frederick P. Brooks, Jr.: The Mythical Man-Month. Essays on Software Engineering. Anniversary Edition. Addison-Wesley, 1995.

Avons-nous perdu le Web que nous aimions ?

Conflit de génération sur le Web…

Les pères fondateurs avaient imaginé un réseau ouvert, génératif, bidouillable.

Ils sont aujourd'hui amers de constater que le Web est devenu un adolescent qui loin de chercher à émanciper ses utilisateurs, tente plutôt de les forcer dans des cases, de les infantiliser, de ne leur laisser aucun contrôle.

Le constat dressé la semaine dernière par Anil Dash a depuis

été largement partagé par les vétérans du Web. Comme l'impression d'un paradis perdu.

Mais la roue tourne et les utopistes des débuts rêvent d'un retour aux sources, d'éduquer les milliards de nouveaux internautes, de leur faire partager leur rêve. Est-ce envisageable ? Surtout, même s'ils prenaient conscience des valeurs que portait le Web à ses débuts, la majorité des utilisateurs serait-elle prête à abandonner ses usages confortables actuels pour reprendre le flambeau des fondateurs et à explorer de nouvelles voies respectueuses de ces valeurs ?

Le retour au bricolage high-tech avec un fer à souder allié au code (Ardhuino, imprimantes 3D, FabLabs…), les initiatives comme celles du projet Webmakers qui vise à éduquer au Web toute une génération pour qu'elle s'en empare au lieu de le consommer, autant de signes d'une prise de conscience qui pourrait modifier la donne. Cet article qui lance un coup d'œil dans le rétroviseur n'est pas un moment de simple nostalgie mais une invitation au renouvellement des idéaux fondateurs.

Le Web que nous avons perdu

article original The Web We Lost par Anil Dash, proposé et présenté par Clochix

Traduction framalang Zii, KoS, Goofy, Garburst, lamessen

L'industrie technologique et sa presse ont traité l'explosion des réseaux sociaux et l'omniprésence des applications pour smartphone comme une victoire sans appel pour Monsieur Tout-le-monde, un triomphe de la convivialité et de l'autonomisation. On a moins parlé de ce que nous avons perdu tout au long de cette transition, et je trouve que les jeunes générations ne savent même pas comment était le Web autrefois. Alors voici quelques aperçus d'un Web qui pour l'essentiel a

disparu :

- Il y a 5 ans, la plupart des photos qu'on voulait partager étaient chargées sur Flickr, où elles pouvaient être taguées par les humains ou même par les applications et services, en utilisant un système de balises. Les images étaient facilement accessibles sur le Web, en utilisant de simples flux RSS. Et les photos chargées pouvaient facilement l'être sous des licences permissives comme celles fournies par Creative Commons, autorisant la modifications et la réutilisation de n'importe quelle façon par des artistes, des entreprises ou des particuliers.
- Il y a une dizaine d'années, Technorati vous laissait chercher sur la majeure partie du Web social en temps réel (cependant la recherche avait tendance à être horriblement longue pour l'affichage des résultats), avec des tags qui marchaient comment le font les hashtags sur Twitter aujourd'hui. Vous pouviez trouver des sites en relation avec votre contenu avec une simple recherche, et savoir qui s'exprimait dans un fil de discussion, indépendamment des outils ou plateformes utilisés pour exposer des idées. À l'époque, c'était tellement excitant que lorsque Technorati ne put faire face au volume croissant de la blogosphère, les utilisateurs furent très déçus. Au point que même quelqu'un d'aussi habituellement circonspect que Jason Kottke se mit à descendre le service en flammes pour l'avoir laissé tomber. Dès l'instant de ses premiers succès pourtant, Technorati avait suscité les louanges de gens comme John Gruber :

Vous pouviez, en théorie, écrire un logiciel pour examiner le code source des quelques centaines de milliers de blogs, et créer une base de données des liens entre ces blogs. Si votre logiciel était assez efficace, il devait pouvoir rafraîchir ses informations d'heure en heure, ajoutant de nouveaux liens

à sa base de données en quasi-temps réel. En fait, c'est exactement ce qu'a créé Dave Sifry avec son incroyable Technorati. À ce jour, Technorati surveille 375 000 blogs et a référencé plus de 38 millions de liens. Si vous n'avez jamais joué avec Technorati, vous manquez quelque chose.

- Il y a dix ans, vous pouviez laisser les gens poster des liens sur votre site ou montrer des listes de liens qui pointaient vers votre site. Car Google n'avait pas encore introduit AdWords et AdSense, les liens ne généraient pas de revenus, c'était juste un moyen d'expression ou un outil éditorial. Le Web était un endroit intéressant et différent avant la monétisation des liens, mais en 2007 il devint clair que Google avait changé le Web pour toujours, et pour le pire, en corrompant les liens.
- En 2003, si vous aviez introduit un service d'authentification individuelle opéré par une société, même en documentant le protocole et encourageant les autres à cloner le service, vous auriez été décrit comme quelqu'un qui introduisait un système de surveillance relevant du « Patriot Act ». Il y avait une telle méfiance à l'égard services d'authentification que même Microsoft abandonna ses tentatives de créer un tel système d'inscription. Même si leur expérience utilisateur n'était pas aussi simple que la possibilité omniprésente de s'identifier avec Facebook ou Twitter, le service TypeKey introduit alors avait des conditions légales de partage des données bien plus restrictives. Et pratiquement tous les systèmes qui fournissaient une identité aux utilisateurs autorisaient l'usage de pseudonymes, respectant le besoin qu'ont les gens de ne pas toujours se servir de leur identité légale.
- Au début de ce siècle, si vous aviez créé un service qui permettait aux utilisateurs de créer ou partager du contenu, ils s'attendaient à pouvoir facilement

télécharger une copie fidèle de leurs données, ou importer leurs données vers d'autres services compétitifs, sans la moindre restriction. Les entreprises commerciales passaient des années à travailler sur l'interopérabilité autour des échanges de données simplement pour le bénéfice de leurs utilisateurs, quitte à lever théoriquement les barrières pour l'entrée de la concurrence.

- Aux premiers temps du Web social, il était largement admis que de simples gens pourraient être propriétaires de leur identité en ayant leurs propres sites, plutôt que d'être dépendants de quelques gros sites pour héberger leur identité numérique. Dans cette vision, vous possédez votre nom de domaine et contrôlez complètement son contenu, plutôt que d'avoir les mains liées sur un site géré par une grande entreprise. C'était une réaction sensée lorsqu'on prenait conscience que la popularité des gros sites croît et chute, mais que les gens ont besoin d'une identité plus persistante que ces sites.

- Il y a cinq ans, si vous vouliez publier sur votre site du contenu d'un autre site ou d'une application, vous pouviez le faire en utilisant un format simple et documenté, sans avoir besoin de négocier un partenariat ou un accord contractuel entre les sites. L'expérience des utilisateurs n'était donc pas soumise aux caprices des luttes politiques entre les sociétés, mais basée sur l'architecture extensible du Web lui-même.

- Il y a une douzaine d'années, lorsque les gens voulaient soutenir les outils de publication qui symbolisaient cet état d'esprit, ils mutualisaient le coût des serveurs et des technologies nécessaires à ces outils, même si cela coûtait bien plus cher avant l'avènement de l'informatique *dans le nuage* et la baisse du prix de la bande passante. Leurs pairs de l'univers des

> technologies, même s'ils étaient concurrents, participaient même à cet effort.

Ce n'est pas notre Web aujourd'hui. Nous avons perdu les éléments-clés auxquels nous faisions confiance et pire encore, nous avons abandonné les valeurs initiales qui étaient le fondement du monde du Web. Au crédit des réseaux sociaux actuels, ils ont apporté des centaines de millions de nouveaux participants sur ces réseaux, et ils ont sans doute rendu riche une poignée de personnes. Mais ils n'ont pas montré le Web lui-même, le respect et l'attention qu'il mérite comme le support qui leur a permis de réussir. Et ils sont maintenant en train de réduire les possibilité du Web pour une génération entière d'utilisateurs qui ne comprennent pas à quel point leur expérience pourrait être beaucoup plus innovante et significative.

Retour vers le futur

Aujourd'hui, lorsque vous voyez des compilations intéressantes d'informations, elles utilisent encore souvent des photos de Flickr, parce qu'il n'y a pas grand-chose à faire avec les maigres métadonnées d'Instagram, et qu'Instagram n'utilise le Web qu'à contre-cœur. Lorsque nous ne pouvons pas retrouver d'anciens messages sur Twitter ou nos propres publications sur Facebook, nous trouvons des excuses aux sites alors que nous avions de meilleurs résultats avec une recherche sur Technorati, qui n'avait portant à sa disposition que de piètres logiciels de son époque. Nous assistons à de stupides combats de coqs avec Tumblr qui ne peut pas récupérer la liste de vos contacts sur Twitter, ou Facebook qui refuse que les photos d'Instagram s'affichent sur Twitter, tout cela parce que des entreprises géantes suivent chacune leur propre programme de développement au lieu de collaborer pour être utiles aux utilisateurs. Et nous nous coltinons une génération de patrons qui sont incités à créer des produits toujours plus bornés et hostiles au Web, tout cela pour permettre à un petit

nombre de nantis de devenir toujours plus riches, au lieu de laisser les gens se créer de nouveaux possibles innovantes au dessus du Web lui-même.

Je ne m'inquiète pas, nous allons corriger tout cela. L'industrie technologique, comme toutes les industries, suit des cycles, et le pendule est en train de revenir vers les philosophies globales et émancipatrices sur lesquelles le Web social s'est bâti au début. Mais nous allons devoir affronter un gros défi, ré-éduquer un milliard d'utilisateurs pour leur apprendre ce qu'est le Web, comme nous l'avons fait pendant des années il y a dix ans quand tout le monde a quitté AOL, leur apprendre qu'il y a bien plus à expérimenter sur Internet que ce qu'ils connaissent.

Ce n'est pas ici la polémique habituelle à base de : « ces réseaux verrouillés sont mauvais ». Je sais que Facebook, Twitter, Pinterest, LinkedIn et tous les autres sont de super-sites, qui apportent beaucoup à leurs utilisateurs. D'un point de vue purement logiciel, ce sont de magnifiques réussites. Mais ils sont basés sur quelques hypothèses qui ne sont pas forcément exactes. La première idée fausse d'où découlent beaucoup de leurs erreurs est que donner aux utilisateurs de la flexibilité et du contrôle crée forcément une expérience utilisateur complexe qui empêche leur croissance. La seconde hypothèse erronée, plus grave encore, est de penser qu'exercer un contrôle total sur les utilisateurs est le meilleur moyen de maximiser les profits et la rentabilité de leur réseau.

La première étape pour les détromper, c'est que les gens qui sont en train de créer la prochaine génération d'applications sociales apprennent un peu d'histoire, pour savoir de quoi ils parlent, qu'il s'agisse du modèle économique de Twitter, des fonctions sociales de Google ou de n'importe quoi d'autre. Nous devons savoir ce qui a été essayé et a échoué, quelles bonnes idées étaient tout simplement en avance sur leur temps, et quelles occasions ont été gâchées par la génération actuelle de réseaux sociaux dominants.

Qu’est-ce que j’oublie ? Qu’avons-nous perdu d’autre sur le Web social ?

Tous les autres pourraient avoir tort, mais c’est peu probable (Libres conseils 2/42)

Tous les autres pourraient avoir tort, mais c’est peu probable

par Evan Prodromou

Evan Prodromou est le fondateur de Wikitravel, StatusNet et du réseau social Open Source Identi.ca. Il participe aux logiciels Open Source depuis 15 ans en tant que développeur, écrit de la documentation et se distingue à l’occasion comme agitateur. Il vit au Québec, à Montréal.

La plus importante caractéristique du fondateur d’un projet Open Source, dans les premières semaines ou premiers mois avant de lancer son logiciel dans le monde, c’est une obstination de tête de mule face à l’écrasante évidence des faits. Si votre logiciel est si important, pourquoi personne ne l’a-t-il déjà écrit ? Peut-être que ce n’est même pas possible. Peut-être que personne d’autre que vous ne veut ce que vous êtes en train de faire. Peut-être que vous n’êtes pas assez bon pour le faire. Peut-être que quelqu’un l’a déjà fait et que vous n’êtes simplement pas assez malin pour le trouver avec Google.

Garder la foi à travers cette longue et sombre nuit est difficile ; seules les têtes de cochons opiniâtres et bornées peuvent y parvenir. Et nous arrivons à l'application de nos opinions de programmeur les plus fortement défendues. Quel est le meilleur langage de programmation à utiliser ? L'architecture de l'application ? Les standards d'écriture du code ? La licence logicielle ? Le système de gestion de version ? Si vous êtes le seul à travailler (ou à connaître !) le projet, vous devez décider, de façon unilatérale.

Quand vous vous lancez finalement, malgré tout, cette détermination bornée et cette forte opinion sont devenus préjudiciables, pas bénéfiques. Une fois que vous vous êtes lancé, vous aurez besoin de compétences tout à fait à l'opposé pour faire des compromis afin que votre logiciel soit davantage utile aux autres. Et beaucoup de ces compromis vous sembleront vraiment mauvais.

Il est difficile de recevoir des avis d'« étrangers » (c.à.d. des personnes qui ne sont pas vous). D'abord parce qu'ils se focalisent sur des choses si triviales, sans importance (votre convention de nommage des variables par exemple, ou l'emplacement de certain boutons). Ensuite parce qu'ils ont invariablement tort . Après tout, si ce que vous avez fait n'est pas la bonne manière de faire, vous ne l'auriez pas fait ainsi dès le départ. Si votre façon de faire n'était pas la bonne, pourquoi votre code serait si populaire ?

Mais « mauvais » est relatif. Si faire un *mauvais* choix rend votre logiciel plus accessible aux utilisateurs finaux, ou aux « développeurs en aval » ou aux administrateurs ou les empaqueteurs, est-ce que ce n'est pas finalement juste ?

Et la nature de ce genre de commentaires et de contributions est généralement négative. Les retours de la communauté sont principalement des réactions, ce qui implique qu'elles sont critiques. Avez vous déjà rapporté un bug qui disait « J'aime

beaucoup l'organisation du module hashtable.c » ou « Bravo d'avoir supprimé ce sous-sous-sous-menu » ? Les gens font un retour d'expérience car ils n'aiment pas la façon dont fonctionne votre logiciel à un instant T. Et ils ne sont pas toujours très diplomates à ce moment-là.

Il est difficile de répondre de façon positive à ce genre de retour. Nous engueulons parfois les expéditeurs sur nos listes des discussions de développement, ou fermons les rapports d'anomalies avec un rictus et un WONTFIX (NdT : NESERAPASRESOLU, employé dans les rapports de bug). Pire encore, nous nous retirons dans notre cocon, ignorant les suggestions externes ou les retours d'expérience, câlinant notre confortable code qui sied parfaitement à nos idées préconçues et à nos marottes.

Si le logiciel n'est que pour vous-même, vous pouvez garder le code source et les infrastructures qui l'entourent comme terrain de jeu personnel. Mais si vous voulez que votre logiciel soit utilisé, qu'il compte pour les autres personnes, qu'il change (peut-être) le monde, alors vous allez devoir construire une saine et solide communauté d'utilisateurs, de contributeurs principaux, d'administrateurs et de développeurs de modules. Les utilisateurs ont besoin d'avoir l'impression de posséder le logiciel, de la même façon que vous.

Il est difficile de se rappeler que chacune de ces voix dissidentes ne représente qu'une infime minorité. Imaginez tous les gens qui entendent parler de votre logiciel qui ne prennent jamais le temps de l'essayer. Ceux qui le téléchargent mais ne l'installent jamais. Ceux qui l'installent, restent bloqués, et l'abandonnent en silence. Et ceux qui veulent vous faire un retour, mais qui ne trouvent pas votre système de rapport de bugs, listes de mails de développeurs, canaux IRC ou adresses mails personnelles. Étant donné les difficultés à faire passer leur message, il y a probablement une centaine de personnes qui veulent que des modifications soient faites pour une seule qui parvient à

transmettre le message. Donc, être à l'écoute de ces voix, quand elle parviennent à vous atteindre, est essentiel.

Le responsable de projet est chargé de maintenir la vision et la finalité du logiciel. Nous ne pouvons vaciller, en faisant des allers et retours basés sur tel ou tel courriel d'utilisateur pris au hasard. Et s'il y a un principe de base en jeu, alors, bien sûr, il est important de garder cette base stable. Personne d'autre que le responsable de projet ne peut le faire.

Mais nous devons réfléchir : y a-t-il des questions non fondamentales qui puissent rendre le logiciel plus accessible, plus facile d'utilisation ? Finalement, la mesure de notre travail est dans la façon dont nous touchons les utilisateurs, comment notre logiciel est utilisé, et ce pourquoi il est utilisé. À quel point notre idée personnelle importe-t-elle « vraiment » pour le projet et pour la communauté ? Quelle part est uniquement ce que le responsable aime, personnellement ? Si ces problèmes non essentiels existent, alors il faut réduire les désaccords, répondre aux demandes, et faire les changements. Le projet n'en sera que meilleur pour tout le monde.

Tous les autres pourraient avoir tort, mais c'est peu probable

par Evan Prodromou

Evan Prodromou est le fondateur de Wikitravel, StatusNet et du réseau social Open Source Identi.ca. Il participe aux logiciels Open Source depuis 15 ans en tant que développeur, écrit de la documentation et se distingue à l'occasion comme agitateur. Il vit au Québec, à Montréal.

La plus importante caractéristique du fondateur d'un projet Open Source, dans les premières semaines ou premiers mois avant de lancer son logiciel dans le monde, c'est une

obstination de tête de mule face à l'écrasante évidence des faits. Si votre logiciel est si important, pourquoi personne ne l'a-t-il déjà écrit ? Peut-être que ce n'est même pas possible. Peut-être que personne d'autre que vous ne veut ce que vous êtes en train de faire. Peut-être que vous n'êtes pas assez bon pour le faire. Peut-être que quelqu'un l'a déjà fait et que vous n'êtes simplement pas assez malin pour le trouver avec Google.

Garder la foi à travers cette longue et sombre nuit est difficile ; seules les têtes de cochons opiniâtres et bornées peuvent y parvenir. Et nous arrivons à l'application de nos opinions de programmeur les plus fortement défendues. Quel est le meilleur langage de programmation à utiliser ? L'architecture de l'application ? Les standards d'écriture du code ? La licence logicielle ? Le système de gestion de version ? Si vous êtes le seul à travailler (ou à connaître !) le projet, vous devez décider, de façon unilatérale.

Quand vous vous lancez finalement, malgré tout, cette détermination bornée et cette forte opinion sont devenus préjudiciables, pas bénéfiques. Une fois que vous vous êtes lancé, vous aurez besoin de compétences tout à fait à l'opposé pour faire des compromis afin que votre logiciel soit davantage utile aux autres. Et beaucoup de ces compromis vous sembleront vraiment mauvais.

Il est difficile de recevoir des avis d'« étrangers » (c.à.d. des personnes qui ne sont pas vous). D'abord parce qu'ils se focalisent sur des choses si triviales, sans importance (votre convention de nommage des variables par exemple, ou l'emplacement de certain boutons). Ensuite parce qu'ils ont invariablement tort . Après tout, si ce que vous avez fait n'est pas la bonne manière de faire, vous ne l'auriez pas fait ainsi dès le départ. Si votre façon de faire n'était pas la bonne, pourquoi votre code serait si populaire ?

Mais « mauvais » est relatif. Si faire un *mauvais* choix rend votre logiciel plus accessible aux utilisateurs finaux, ou aux « développeurs en aval » ou aux administrateurs ou les empaqueteurs, est-ce que ce n'est pas finalement juste ?

Et la nature de ce genre de commentaires et de contributions est généralement négative. Les retours de la communauté sont principalement des réactions, ce qui implique qu'elles sont critiques. Avez vous déjà rapporté un bug qui disait « J'aime beaucoup l'organisation du module hashtable.c » ou « Bravo d'avoir supprimé ce sous-sous-sous-menu » ? Les gens font un retour d'expérience car ils n'aiment pas la façon dont fonctionne votre logiciel à un instant T. Et ils ne sont pas toujours très diplomates à ce moment-là.

Il est difficile de répondre de façon positive à ce genre de retour. Nous engueulons parfois les expéditeurs sur nos listes des discussions de développement, ou fermons les rapports d'anomalies avec un rictus et un WONTFIX (NdT : NESERAPASRESOLU, employé dans les rapports de bug). Pire encore, nous nous retirons dans notre cocon, ignorant les suggestions externes ou les retours d'expérience, câlinant notre confortable code qui sied parfaitement à nos idées préconçues et à nos marottes.

Si le logiciel n'est que pour vous-même, vous pouvez garder le code source et les infrastructures qui l'entourent comme terrain de jeu personnel. Mais si vous voulez que votre logiciel soit utilisé, qu'il compte pour les autres personnes, qu'il change (peut-être) le monde, alors vous allez devoir construire une saine et solide communauté d'utilisateurs, de contributeurs principaux, d'administrateurs et de développeurs de modules. Les utilisateurs ont besoin d'avoir l'impression de posséder le logiciel, de la même façon que vous.

Il est difficile de se rappeler que chacune de ces voix dissidentes ne représente qu'une infime minorité. Imaginez tous les gens qui entendent parler de votre logiciel qui ne

prennent jamais le temps de l'essayer. Ceux qui le téléchargent mais ne l'installent jamais. Ceux qui l'installent, restent bloqués, et l'abandonnent en silence. Et ceux qui veulent vous faire un retour, mais qui ne trouvent pas votre système de rapport de bugs, listes de mails de développeurs, canaux IRC ou adresses mails personnelles. Étant donné les difficultés à faire passer leur message, il y a probablement une centaine de personnes qui veulent que des modifications soient faites pour une seule qui parvient à transmettre le message. Donc, être à l'écoute de ces voix, quand elle parviennent à vous atteindre, est essentiel.

Le responsable de projet est chargé de maintenir la vision et la finalité du logiciel. Nous ne pouvons vaciller, en faisant des allers et retours basés sur tel ou tel courriel d'utilisateur pris au hasard. Et s'il y a un principe de base en jeu, alors, bien sûr, il est important de garder cette base stable. Personne d'autre que le responsable de projet ne peut le faire.

Mais nous devons réfléchir : y a-t-il des questions non fondamentales qui puissent rendre le logiciel plus accessible, plus facile d'utilisation ? Finalement, la mesure de notre travail est dans la façon dont nous touchons les utilisateurs, comment notre logiciel est utilisé, et ce pourquoi il est utilisé. À quel point notre idée personnelle importe-t-elle « vraiment » pour le projet et pour la communauté ? Quelle part est uniquement ce que le responsable aime, personnellement ? Si ces problèmes non essentiels existent, alors il faut réduire les désaccords, répondre aux demandes, et faire les changements. Le projet n'en sera que meilleur pour tout le monde.