UNIVERSITÉ DU QUÉBEC À MONTRÉAL

ET

TÉLÉCOM PARISTECH

LE CODE SOURCE INFORMATIQUE COMME ARTEFACT DANS LES RECONFIGURATIONS D'INTERNET

THÈSE

PRÉSENTÉE

COMME EXIGENCE PARTIELLE

DU DOCTORAT EN COMMUNICATION (UQAM)

ET DU DOCTORAT EN SCIENCES ÉCONOMIQUES ET SOCIALES

(TÉLECOM PARISTECH)

PAR

STÉPHANE COUTURE

DÉCEMBRE 2012

AVANT-PROPOS

> Notre conviction sous-jacente à cette approche est que l'informatique n'est pas une science et que sa signification n'a pas grand-chose à voir avec les ordinateurs. La révolution informatique est une révolution dans le mode de pensée et dans l'expression de celle-ci (Abelson et Sussman, 1996, p. xviii)

C'est en 1996, alors que je poursuivais mes études de premier cycle universitaire en informatique, que ce passage provocateur a capté mon attention. Il s'agissait d'un extrait d'un manuel scolaire utilisé dans l'enseignement de Lisp, un vieux langage de programmation à la syntaxe particulière. Cet extrait m'avait alors frappé car il rejoignait mon sentiment que la programmation informatique n'était pas qu'un outil pour créer des applications informatiques, mais qu'elle pouvait être également une activité abordée en soi, comme une créatrice de sens, au même titre qu'une production littéraire ou une œuvre musicale, par exemple. Depuis, au fil de mes recherches et déambulations, j'ai pu constater que cette façon de considérer la dimension « expressive » de la programmation n'est pas isolée. Au début de mes études de doctorat, j'ai ainsi pris connaissance d'une communication réalisée par Donald Knuth, auteur d'un ouvrage (Knuth, 1968) considéré comme l'une des monographies scientifiques les plus riches du 20e siècle (Morrison et Morrison, 1999), qui insiste sur l'importance du style dans la programmation, et sur la nécessité pour les programmeurs de trouver le meilleur style pour s'exprimer (Knuth, 1974, p. 670). De façon plus contemporaine, les militants du logiciels libres, inspirés en cela par les auteurs cités plus tôt, mettent désormais de l'avant l'idée que la programmation informatique est une forme d'expression et que l'accès au code source est un droit humain. Rappelons qu'un logiciel peut être considéré comme étant libre – ou « open source » – s'il est possible de partager, voire améliorer ou redistribuer son code source, sans aucune restriction.

C'est dans ce sens, à mon avis, que l'on peut le mieux saisir la pertinence, sinon l'importance d'appréhender la programmation informatique dans une perspective communicationnelle : en tant que cet artefact permet ou facilite la communication entre humains. Toutefois, il est remarquable de ne trouver que très peu d'études consacrées à cette dimension en communication ou en sciences sociales. En effet, bien que de nombreuses études aient été

consacrées à différents aspects des technologies de l'information et en particulier aux logiciels libres, il est étonnant que très peu d'études ne soient consacrées au *code source*, pourtant l'objet de la programmation informatique et l'enjeu central de la lutte des militants des logiciels libres. Une telle situation m'apparaît semblable à celle, surréaliste, qui consisterait à s'intéresser à la diffusion des livres, à leur partage entre les acteurs, sans jamais *ouvrir* un livre, sans décrire la manière dont il est lu.

C'est dans cette perspective que la présente thèse est consacrée au *code source,* cet artefact qui est en quelque sorte l'objet de la programmation informatique, objet qui prend une dimension culturelle et politique grandissante aujourd'hui. Mon principal objectif est donc, bien modestement, de proposer une étude qui se penche de façon frontale sur la question : *qu'est-ce que le code source ?* Je cherche ensuite à saisir la dimension sociale et communicationnelle du code source, en m'attardant au travail de transformation et de fabrication de cet artefact. Si d'autres travaux ont effectivement abordé la collaboration dans l'univers des logiciels libres, la contribution de mon étude se situe dans l'articulation plus étroite que je fais entre l'analyse de cette collaboration et celle de l'artefact code source.

Mentionnons ici que l'une des préoccupations initiales de la thèse concernait la question de la « beauté du code ». Cette question était à ce point importante que j'ai réalisé un travail de session portant ce titre. J'ai cependant ensuite réduit la place qu'occupait la beauté pour plutôt me concentrer sur ce que j'appelais les « appréciations » du code source, soit les différentes manières de juger ou d'apprécier le code source. Dans la suite de l'étude, cette place accordée à la beauté a été encore davantage réduite pour explorer d'autres aspects. Il me semblait par exemple important de documenter ce que constituait le code source, d'une part et d'autre part, de ne pas oublier la dimension politique du code source, a priori difficile à saisir par la seule question de la « beauté » et des appréciations. Le chapitre 5 tente toutefois d'approfondir la question de la « beauté », en analysant le discours des acteurs à ce propos et en faisant ressortir quelques implications possibles de cette présumée « beauté du code », en termes de design et de collaborations entre humains et machines. Si les analyses présentées dans ce chapitre pourraient sans doute être poussées davantage, c'est celui qui aborde à mon sens les enjeux les plus fondamentaux de la thèse, tant en termes politiques qu'anthropologiques.

Le plus difficile dans cette thèse aura été de la terminer et par conséquent, d'accepter qu'elle ne soit pas parfaite. Dans ce sens, le lecteur ou la lectrice pourra certainement y retrouver de nombreuses imperfections de forme, voire certaines lacunes de fond. J'espère néanmoins que cette thèse aura le mérite de proposer des pistes permettant d'aborder dans toute sa complexité la question suivante : *qu'est-ce que le code source ?*

Remerciements

J'aimerais tout d'abord remercier les acteurs de symfony et de SPIP qui ont accepté de participer à cette étude, soit en m'accordant quelques heures de leur temps ou bien simplement en discutant avec moi.

J'aimerais ensuite remercier mes directeurs de thèse. D'abord, Serge Proulx, qui m'a suivi depuis les sept dernières années, à la maîtrise et au doctorat, et qui m'a toujours soutenu et fait confiance, même dans les moments plus difficiles. Merci également à Christian Licoppe, pour avoir accepté de co-diriger ma thèse en cotutelle, ainsi que pour son insistance à ce que j'oriente celle-ci vers l'étude du code source. Je remercie également les membres du jury qui ont pris le temps de lire ma thèse et la commenter.

J'aimerais également remercier les professeur-es et chercheur-es qui m'ont accompagné à différents moments dans mon parcours. Au doctorat conjoint en communication : : Gaétan Tremblay, Florence Millerand et Éric George de l'UQAM, Leslie Regan Shade (alors à Concordia) et Monika Kin Gagnon, Lorna Heaton et François Cooren, à l'Université de Montréal. Merci à Jérôme Denis et Françoise Détienne à Télécom ParisTech, et finalement, à Robert Dupuis, du département d'informatique de l'UQAM pour son soutien et nos discussions. Je remercie également mes collègues étudiant-es pour nos discussions, en particulier ceux et celles que j'ai côtoyés durant les dix dernières années (!) au Laboratoire de communication médiatisée par ordinateurs. Je tiens particulièrement à remercier Anne Goldenberg, ma grande amie, de même que Guillaume Latzko-Toth, Sylvie Jochems et Christina Haralanova avec qui j'ai pu interagir de façon plus étroite.

Merci également à mes parents et amis qui m'ont soutenu durant cette thèse. D'abord, merci à ma mère Francine Grenier et mon père Gaston Couture, pour leur soutien de tous les jours.

J'aimerais également remercier quelques amis qui m'ont appuyé dans mon parcours. Merci en particulier à Antoine Beaupré. Nos débats, souvent enflammés et parfois douloureux, ont toujours été une riche source d'inspiration intellectuelle. Olivier Loyer, dont l'énergie et le goût du plein air m'ont souvent permis de décrocher de ma thèse. Je le remercie également, ainsi que François Dubuc et Robin Millette, pour avoir révisé l'un des chapitres de ma thèse. Frédéric Sultan, avec qui j'ai eu plusieurs échanges intellectuels et qui a grandement facilité mon séjour en France. Je remercie d'ailleurs sa conjointe, Sophie Roy-Sultan, ainsi que ses parents (et beaux-parents de Frédéric), François Roy et Bernadette Roy-Jacquey, de m'avoir très généreusement prêté leur appartement dans le 14^{e} arrondissement de Paris durant mon premier séjour de six mois à Paris.

Merci finalement à mon amour, Geneviève Szczepanik, pour ses nombreux commentaires, relectures et révisions, de même que pour son soutien de tous les jours, autant sur le plan personnel que sur le plan intellectuel. Elle fut ma plus précieuse collègue durant toutes ces années.

TABLE DES MATIÈRES

LISTE DES FIGURES

Figure Page

LISTE DES TABLEAUX

RÉSUMÉ

Cette thèse en communication porte sur le code source informatique. Le code source est l'objet de la programmation informatique et peut être défini comme un ensemble de commandes informatiques humainement lisibles, « écrites » dans un langage de programmation de haut niveau (Krysia et Grzesiek, 2008). Depuis quelques années, le code source fait l'objet d'une signification sociale et politique grandissante. Le mouvement du logiciel libre place par exemple au cœur de sa politique le libre accès au code source. Ce mouvement a d'ailleurs permis l'émergence de nombreux collectifs articulés autour de la fabrication collective du code source. Si plusieurs études se sont attardées aux différents aspects de ces collectifs et aux usages des technologies numériques en général, force est toutefois de constater que le code source reste un objet remarquablement négligé dans les études en communication. L'objectif principal de cette thèse est donc d'aborder frontalement l'artefact *code source,* en répondant à cette question centrale de recherche : *qu'est-ce que le code source et comment cet artefact agit-il dans les reconfigurations d'Internet ?*

Notre problématique s'articule selon trois axes. D'abord, le constat de la signification sociale et politique grandissante du code source, qui s'exprime notamment dans un discours faisant du code source une forme d'expression. Ensuite, la manière dont, pour certains auteurs, le code informatique agit à la manière d'une loi en prescrivant ou limitant certains comportements. Finalement, un dernier axe concerne les rapports d'autorité et le travail invisible dans la fabrication du code source. Sur le plan théorique, notre étude se situe à l'intersection du champ « Science, technologie et société » (STS) et de celui des études en communication. Elle s'appuie largement sur les travaux récents de Lucy Suchman (2007) qui cherchent à poser le regard sur des dynamiques de reconfigurations mutuelles et permanentes des relations entre humains et machines. Notre étude mobilise également certains travaux français se situant en continuité de la théorie de l'acteur-réseau, et s'attardant au travail nécessaire à la stabilité et la performativité des artefacts.

D'un point de vue méthodologique, notre étude prend comme terrain *SPIP* et *symfony*, deux logiciels qui ont en commun d'être utilisés comme infrastructures dans le fonctionnement de nombreux sites web interactifs, souvent désignés sous l'appellation « web 2.0 ». Les deux logiciels sont originaires de France et continuent de mobiliser un nombre significatif d'acteurs français. Ces projets se distinguent par les valeurs mises de l'avant, plus militantes et non commerciales dans le cas de *SPIP*, plus professionnelles et commerciales dans le cas de *symfony*. La langue utilisée dans l'écriture du code source est également différente : français pour *SPIP*, anglais pour *symfony*. L'enquête combine l'analyse de documents et de traces en ligne, des entretiens semi-dirigés avec les acteurs des projets, de même que l'observation de différentes rencontres entre les acteurs.

Notre étude fait tout d'abord clairement ressortir une certaine ambiguïté entourant la définition de la notion du « code source ». Alors que le code source est souvent appréhendé comme un « texte », « que l'on écrit », l'analyse des définitions plus formelles, ou encore de l'objet désigné par les acteurs par le terme de « code source », montre que cet objet renvoie souvent à différents types de médias, comme des images, et même des artefacts qui ne sont

pas directement destinés au fonctionnement des ordinateurs. À l'instar des propos de certains acteurs, nous croyons que la définition de ce qui constitue le code source revêt même une dimension politique, dans ce sens qu'elle tend à valoriser certains types d'activités plutôt que d'autres. L'analyse du processus de fabrication collective du code source dans les deux projets montre également des différences importantes au niveau de l'organisation du code source, de même que dans la mise en œuvre des normes et des « autorisations » d'écriture dans chacun des projets. Ces différences s'articulent avec les valeurs des projets et participent d'une certaine configuration du type d'acteur destiné à interagir avec telle ou telle partie du code source.

En conclusion, nous insistons sur le fait que le code source ne doit pas seulement être appréhendé comme étant le noyau des infrastructures d'information. Il doit aussi être appréhendé, dans une perspective communicationnelle et sociologique, comme un artefact à travers duquel des acteurs humains entrent en relation entre eux pour reconfigurer le monde socionumérique au sein duquel ils et elles sont engagés. Suivant l'approche de Suchman, nous proposons donc d'appréhender le code source comme une interface, ou une multiplicité d'interfaces, dans les reconfigurations d'Internet, en insistant sur la manière dont le design de ces interfaces entraîne certaines conséquences, en particulier en privilégiant la participation de certains acteurs plutôt que d'autres.

Mots-clés : code source, logiciels libres, artefacts, études STS, reconfigurations humain-machine.

INTRODUCTION

De nombreuses études ont été consacrées dans les dernières années aux différents aspects sociaux et politiques des technologies de l'information. Parmi ces études, un certain nombre ont également abordé la question des logiciels libres et du code informatique. Elles se sont attachées à étudier diverses dimensions des logiciels libres, dont le principe premier repose sur le libre accès au code source des logiciels. Ces études ont par exemple abordé les valeurs hackers des acteurs impliqués dans les communautés de logiciels libres (Auray, 2000, 2009) les formes de régulation qui assurent la cohésion de ces communautés (Demazière, Horn, et Zune, 2006, 2007), ou encore l'articulation entre la production du logiciel libre et le capitalisme contemporain (Coris, 2006). D'autres études, s'inspirant des discours des militants du logiciel libre, ont insisté sur la manière dont le code informatique constitue une forme d'écriture ou une forme expressive (2008) propre à être protégée par le droit à la liberté d'expression (2009) . Enfin, certaines études ont mobilisé la catégorie du « code » pour insister sur le caractère politique des infrastructures technologiques qui agissent à la manière d'une loi en régulant le comportement des usagers (Lessig, 2006; Brousseau et Moatty, 2003). Ainsi, Brousseau et Moatty écrivent que « l'écriture de lignes de code informatique commandant l'exécution de procédures revient à imposer aux utilisateurs des normes d'usages » (Brousseau et Moatty, 2003, p. 8).

Si ces études proposent des réflexions souvent fort pertinentes, force est de constater que l'artefact code source y occupe la plupart du temps une place tout à fait périphérique, s'il n'est pas simplement absent. Ainsi, si beaucoup d'études abordent les différents aspects d'Internet et que la notion de « code » revient parfois pour analyser les dispositifs techniques, le « code source » reste encore un objet particulièrement, voire totalement, ignoré dans ces études. Même dans le cas des nombreuses études sur les logiciels libres, si le terme « code source » est parfois mentionné, ce n'est surtout que de manière périphérique. Par exemple, dans un article où ils analysent les relations de travail dans la communauté SPIP, Demazière, Horn et Zune (2007a) n'accordent que deux courtes notes aux notions de « code », qu'ils définissent comme « le texte des instructions du logiciel, écrit dans un langage de programmation » (p. 103) et le codeur comme celui qui « écrit le texte du logiciel, i.e. les instructions du

programme informatique » (p. 105). La notion de « code source » est cependant absente de leur analyse. De la même manière, si l'ouvrage de Doueihi (2008) compare explicitement le code à une forme d'écriture, il ne présente aucun extrait, capture d'écran ou description précise de ce qui constitue concrètement le code source et son processus d'écriture[1]. Malgré leur signification sociale et culturelle grandissante et leur importance cruciale comme composantes des technologies numériques, des notions telles que « logiciel », « code » ou « code source » restent encore très peu problématisées en sciences sociales (Rooksby et Martin, 2006; Fuller, 2008). Très rares sont les études, en sciences sociales, qui s'interrogent aujourd'hui sur ce que constitue exactement le code source informatique et la manière dont les humains font usage, transforment et fabriquent cet artefact. Une exception notable à cet égard est le courant émergent des *code studies*, dont l'objectif semble être d'analyser la place grandissante du « code », sous toutes ses formes, dans nos sociétés. Cependant, même dans ces études, le « code source » reste assez peu problématisé, malgré l'importance grandissante de cette notion, même à l'extérieur du monde de l'informatique. Notons par exemple la sortie récente d'un roman de fiction et d'un film portant ce titre, alors que leur objet n'a absolument rien à voir avec celui que nous analysons dans cette thèse.

L'objectif principal de cette thèse est d'aborder frontalement l'objet « code source », en documentant avec précision les définitions que certains concepteurs de logiciels libres donnent à la notion de code source et à décrivant le processus de fabrication collective de cet artefact. Notre démarche s'appuie principalement sur l'étude de deux projets de logiciels libres : SPIP et symfony. Ces deux logiciels libres ont en commun d'être écrits dans le langage informatique PHP, et d'agir comme une infrastructure pour de nombreux sites web interactifs souvent désignés par l'appellation de web 2.0. Ce que nous analysons, c'est donc en quelque sorte le code source du web social.

La thèse comporte sept chapitres en incluant la conclusion générale. Les trois premiers chapitres présentent respectivement la problématique, le cadre théorique et l'approche méthodologique qui guide la thèse. Le premier chapitre est consacré à la spécification de notre problématique, qui s'articule au constat d'une signification sociale et politique

1 Mentionnons toutefois que l'auteur présente une capture d'écran du site ohloh.net offrant quelques statistiques sur le code source du noyau Linux (Doueihi, 2008, ill. 16).

grandissante attribuée au code source informatique. Nous faisons une première revue de différents travaux ayant abordé la question du code source, en particulier en tant qu'il constitue une forme d'écriture, et qu'il agit à la manière d'une loi. La deuxième partie tente de préciser davantage une première définition du « code source informatique ». Nous remarquons que la définition du code source, loin d'être bien établie, est plutôt instable et en voie de formalisation. Finalement, nous positionnons davantage notre objet d'étude à partir d'une revue de différents travaux en sciences sociales ayant abordé de façon plus ou moins directe ou périphérique, l'artefact « code source ».

Le second chapitre présente différents travaux qui ont alimenté notre réflexion théorique. Il s'agit en quelque sorte de présenter la posture théorique, voire philosophique, à partir de laquelle nous réalisons nos analyses. Le fil rouge de ce développement théorique est à situer dans le développement que Suchman propose dans son ouvrage *Human-Machines Reconfigurations (2007)* et consiste à considérer le travail d'assemblage et de reconfiguration sociotechnique, et en particulier le travail de fabrication du code source informatique, comme un projet social et culturel, qui est continuellement remis en question. Plus spécifiquement, nos concepts s'appuient sur différents travaux en sociologie des sciences et des techniques, de même qu'en anthropologie de l'écriture, qui ont commun de s'intéresser à la capacité d'action des artefacts.

Le troisième chapitre présente notre démarche méthodologique. Comme indiqué plus tôt, notre approche méthodologique s'appuie sur l'étude de la fabrication du code source, dans le cadre de deux projets de logiciels libres, SPIP et symfony. Après avoir présenté les grands principes de notre approche méthodologique, le chapitre présente plus en détails les projets étudiés, ainsi que les techniques d'enquêtes mobilisées. Celles-ci combinent l'analyse de documents et de traces en ligne, des entretiens semi-dirigés avec les acteurs des projets, de même que l'observation lors de différentes rencontres entre les acteurs.

La seconde partie de la thèse, qui comprend les chapitres 4, 5 et 6, constitue l'analyse de nos données empiriques. L'objectif du chapitre 4 est de rendre compte des définitions que les acteurs donnent à la notion de code source. La première section analyse plus concrètement l'objet désigné par les termes « code source de symfony », « code source de SPIP » dans les discours des acteurs. Cette analyse fait ressortir un premier découpage dans le code source

des projets étudiés entre, d'une part, le « coeur » (core) du projet et, d'autre part, les « plugins », et montre la circulation du code source dans différents sites. La deuxième section s'attarde aux définitions générales dans les discours des acteurs. Nous faisons ressortir que, bien que la définition « commune » du code source soit généralement partagée, plusieurs ambiguïtés apparaissent à la frontière de la notion. Finalement, en nous appuyant sur les affirmations des acteurs eux-mêmes, sur nos observations, de même que sur la théorie, nous soutenons que la définition même de la notion de « code source » soulève des enjeux politiques, notamment liés à la reconnaissance.

Le chapitre 5 porte sur les normes d'écriture et la qualité du code source. Il prend comme point de départ l'analyse de certaines catégories utilisées par les acteurs pour apprécier la qualité « interne » du code source. Nous abordons la question de la *beauté du code*, qui nous intriguait au début de l'étude, mais qui s'est avérée peu significative dans les discours des acteurs, mais qui nous permet de mettre ensuite en perspective d'autres termes utilisés pour juger de la qualité du code source, telles la propreté, la clarté et plus important encore, la lisibilité. Nous abordons ensuite certaines normes, règles et « bonnes pratiques » qui participent, selon les acteurs, à la qualité du code source. Nous terminons ce chapitre en analysant deux controverses qui mettent en relation différentes catégories de la qualité et une certaine configuration de l'usager (Woolgar, 1991). Ces différentes analyses de la qualité du code renvoient finalement à différentes manières de concevoir *l'interface* du code source, ou plutôt, les différentes interfaces du code source.

Le chapitre 6 décrit l'activité distribuée d'écriture du code source informatique en prenant comme point de départ l'analyse de la gestion des versions du code source et, en particulier, l'acte informatique du « commit ». Le « commit » est une commande informatique implantée dans plusieurs logiciels de gestion des versions de documents électroniques. Dans la première section de ce chapitre, nous décrivons l'usage du logiciel Subversion, un système de gestion des versions du code source, dont le « commit » est la commande centrale. Cette description nous permet notamment d'approfondir notre compréhension de la gestion et de l'organisation du code source. Elle se termine par l'analyse d'une modification précise au code source d'un des projets, modification couronnée par l'acte du commit. La seconde section s'attarde à présenter, dans leurs détails, les droits et les autorisations d'écriture du code source informatique. Nous nous intéressons particulièrement à la manière dont ces droits et

autorisations sont distinctement mis en œuvre pour chacun des projets. La dernière section aborde le logiciel GIT, un logiciel de gestion des versions considéré par les acteurs comme plus adapté à une démarche d'écriture du code source, distribuée et collaborative.

Le fil conducteur de l'ensemble de la thèse consiste à analyser le code source comme un artefact de la reconfiguration continuelle de l'Internet. Dans cette perspective, le code source constituent non seulement un artefact de la conception d'Internet, mais également une interface par lequel les usagers – ou certains types d'usagers – interagissent avec Internet, à travers de nouvelles formes d'écriture et de lecture. Cette perspective rejoint donc celle de Suchman (2007) qui cherche à appréhender de nouvelles formes d'interactions par lesquelles l'humain et la machine se reconfigurent mutuellement. En particulier, la comparaison des projets symfony et SPIP montre que la manière dont sont élaborées les différentes autorisations et conventions dans l'écriture du code source tend à privilégier l'engagement de certains types d'acteurs. Cette perspective ouvre donc la voie sur des nouvelles possibilités d'appréhender le code source d'une manière qui permet à un plus large spectre d'usagers de participer aux reconfigurations sociotechniques.

La contribution de notre thèse se situe au carrefour des études en communication et des études sur les sciences et technologies (STS) et consiste principalement à aborder explicitement la notion de code source. En effet, comme mentionné plus tôt, bon nombre d'études ont été réalisées concernant les multiples aspects des technologies de l'information et des logiciels libres. Cependant, à notre connaissance, aucune ne s'est donné comme objectif d'analyser le code source informatique, un artefact pourtant au cœur de toutes les technologies logicielles. L'intérêt d'adopter cette orientation de recherche est qu'elle permet d'« ouvrir la boîte noire » des médias numériques et d'analyser le noyau des infrastructures technologiques avec lesquelles nous interagissons quotidiennement. Par ailleurs, la manière dont nous avons appréhendé le code source comme un objet d'écriture permet également de contribuer aux travaux réalisés en anthropologie de l'écriture en analysant des formes d'écriture numériques émergentes. Dans cette perspective, il ne s'agit pas simplement d'étudier la fabrication collective d'une technologie. Il s'agit également – et peut-être surtout – de faire ressortir la manière dont la fabrication du code source peut être, en soit, appréhendée comme une pratique d'écriture. Réciproquement, il s'agit de comprendre comment l'écriture et l'organisation du code source influenceront la participation des acteurs. Nous espérons

finalement que notre travail contribuera à des rapprochements interdisciplinaires entre les sciences sociales et le domaine de l'informatique.

CHAPITRE I

LE CODE SOURCE COMME OBJET D'ÉTUDE. PROBLÉMATIQUE, RECENSION DES ÉCRITS ET QUESTIONS DE RECHERCHE

> Now, in a society when everyone who uses a computer is technically adept, you can make a convincing case that having access to software's source code is a human right.
>
> Richard Stallman, *Fondation pour le logiciel libre*[2]

Ce premier chapitre a pour objectif de présenter l'objet de recherche. Il est divisé en quatre parties. La première est consacrée à la spécification de notre problématique, à partir de trois axes différents. D'abord, la signification sociale et politique grandissante du code source, qui s'exprime notamment dans un discours faisant du code source une forme d'expression. Ensuite, la manière dont, pour certains auteurs, le code informatique agit à la manière d'une loi en prescrivant ou limitant certains comportements. Finalement, un dernier axe concerne les rapports d'autorité et le travail invisible dans la fabrication du code source. Suivra ensuite une deuxième partie consacrée à une première revue de différentes définitions de ce qui constitue le code source, et qui amènera à poser le constat que cette définition, loin d'être formellement établie, est encore instable. La troisième partie tente quant à elle de positionner davantage notre objet d'étude à partir d'une recension de différents travaux en sciences sociales ayant abordé, de façon directe ou périphérique, l'artefact « code source ». Ce chapitre se conclura par une dernière partie où nous présenterons plus formellement nos objectifs de recherche et notre question centrale.

---

2 <http://blog.reddit.com/2010/07/rms-ama.html> (consulté le 17 novembre 2011).

1.1 Problématique : la signification sociale et politique grandissante du code source informatique

1.1.1 Prélude : les mouvements des logiciels libres et à « code source ouvert »

Notre questionnement sur le code source prend racine dans les études précédentes auxquelles nous avons participé portant sur les mouvements des logiciels libres et à « code source ouvert » - *open source* - (Couture, 2006; Couture et al., 2010; Proulx, Couture, et Rueff, 2008; Couture, 2007). Le concept de logiciel libre a émergé au début des années 1980 pour faire contrepied aux restrictions à l'utilisation des logiciels qui étaient alors nouvellement imposées par l'industrie naissante d'édition de logiciels. Avant cette période, en effet, le développement des logiciels ne constituait pas un enjeu commercial et le partage des connaissances entre les programmeurs de différentes entreprises ou institutions publiques était la norme. Cette situation change vers la fin des années 1970, alors que les logiciels sont progressivement soumis aux régimes de droits d'auteurs, ce qui a pour conséquence de restreindre les usages et surtout d'interdire à toute fin pratique le partage de leur code source. Face à cette situation, quelques initiatives émergent avec comme objectif, sinon de s'opposer à ces restrictions, du moins d'offrir des solutions de rechange aux logiciels dits « propriétaires » dont l'accès est restreint. L'une des initiatives les plus marquantes est sans doute celle de Richard Stallman, qui s'engage dans la réalisation du projet *GNU,* un système informatique de type Unix, complètement « libre ». Peu de temps après, pour appuyer institutionnellement sa démarche, Stallman crée, conjointement avec d'autres acteurs, la *Fondation pour le logiciel libre* (*Free Software Foundation* – FSF), dont la mission est d'appuyer le développement de logiciels libres. Parallèlement à cette démarche, une définition plus formelle du logiciel libre est publiée, définition qui s'articule selon quatre libertés, numérotées de 0 à 3, et que nous reproduisons ici[3] :

- La liberté d'exécuter le programme, pour tous les usages (liberté 0).
- La liberté d'étudier le fonctionnement du programme, et de l'adapter à vos besoins (liberté 1). Pour ceci l'accès au code source est une condition requise.
- La liberté de redistribuer des copies, donc d'aider votre voisin (liberté 2).

3 <http://www.gnu.org/philosophy/free-sw.fr.html> (consulté le 21 décembre 2011).

- La liberté de distribuer des copies de vos versions modifiées à d'autres (liberté 3). En faisant cela, vous pouvez faire profiter toute la communauté de vos changements. L'accès au code source est une condition requise.

Mentionnons ici que la première motivation de la création de ce mouvement pour les logiciels libres est d'ordre moral. Stallman n'a en effet de cesse de répéter que l'enjeu des logiciels libres n'a pour lui rien à voir avec des enjeux d'efficacité ou de supériorité. Le choix du logiciel est justifié avant tout parce que le système social du logiciel propriétaire est antisocial, non-éthique et simplement mal : « The idea that the proprietary software social system—the system that says you are not allowed to share or change software—is antisocial, that it is unethical, that it is simply wrong, may come as a surprise to some readers » (Stallman, 2004). C'est parce que l'interdiction du partage des logiciels est mal sur le plan moral que les logiciels libres doivent exister. Lors d'une entrevue réalisée avec des journalistes français, Stallman va d'ailleurs aussi loin que d'affirmer que les logiciels libres ne doivent pas être d'abord jugés en fonction de leur efficacité, mais plutôt de leur dimension communautaire :

> Que le logiciel libre aboutisse aussi à du logiciel efficient et puissant a été une surprise pour moi, et je m'en réjouis. Mais c'est un bonus. J'aurais choisi le logiciel libre, même s'il avait été moins efficace et moins puissant—parce que je ne brade pas ma liberté pour de simples questions de convenances (Stallman, 2002).

Au coeur de la définition du logiciel libre, on retrouve donc ici la notion de code source. C'est d'ailleurs, pour la FSF, ce qui distingue le *Freeware* (logiciel gratuit) du *Free Software* (logiciel libre) : ce dernier garantit la « *liberté* » du code source, tandis que dans le premier, seul l'exécutable est *gratuit*[4]. Comme nous le verrons plus loin, les militants du logiciel libre placeront progressivement le code source au coeur de leur argumentaire, en insistant que celui-ci doit être considéré comme une forme d'expression propre à être protégée à l'aune du droit à la liberté d'expression.

Dans les années 1990, le développement de logiciels dits « libres » réussit à mobiliser bon nombre d'informaticiens. Si certains de ces projets sont soutenus par la FSF, d'autres projets

---

4 Note no 9 du manifeste GNU <http://www.gnu.org/gnu/manifesto.html> (consulté le 23 novembre 2011).

évoluent de façon indépendante et les motivations des acteurs sont souvent assez éloignées des préoccupations morales de Stallman et de la *Free Software Foundation*. C'est par exemple le cas de Linux, un projet aujourd'hui bien connu, qui est créé en 1991 par Linus Torvalds, à titre de loisir. Vers la fin des années 1990 et devant la vitalité de certains projets de logiciels libres, en particulier Linux, plusieurs acteurs commencent à percevoir dans les logiciels libres un modèle de développement de logiciel particulièrement adapté aux réseaux de l'Internet et pouvant être supérieur au modèle dit « propriétaire » de développement de logiciel, où l'accès au code source est restreint à un certain nombre d'individus (souvent des employés de la firme éditant le logiciel). L'une des analyses retentissantes de cette période est l'essai *La cathédrale et le bazar* (Raymond, 2001). L'auteur y compare le développement de Linux à un modèle de développement « en bazar » où le coordonnateur du projet puise des morceaux de code source librement accessibles sur Internet – le bazar – pour les assembler dans un projet cohérent. *La cathédrale et le bazar* est par la suite mobilisée par différents acteurs du monde commercial, pour mettre de l'avant la possible supériorité technique, voire économique, d'un modèle de développement de logiciel en réseau, s'appuyant sur l'« ouverture » du code source[5]. C'est à ce moment que l'expression « open source » apparaît, avec pour double objectif d'esquiver le débat moral sur la « liberté » mise de l'avant par Stallman et la *Free Software Foundation*, débat perçu comme rebutant, voire contre-productif, pour la diffusion des logiciels « libres » et « open source » auprès des entreprises commerciales. L'expression « open source » permet de mettre de l'avant un modèle de développement et de diffusion des logiciels basé sur l' « ouverture » des sources, modèle potentiellement supérieur sur les plans techniques et économiques aux logiciels à code source « fermé ». Ce double mouvement du logiciel libre/open source permet également de donner, pour reprendre les termes de Cardon, une cohérence normative et organisationnelle à des processus collaboratifs d'innovations « ascendantes » que l'on retrouve aujourd'hui sur Internet, en particulier sur les wikis (Goldenberg, 2010). Ces principes normatifs et organisationnels permettent le développement de logiciels libres aujourd'hui très populaires comme Linux, Firefox ou OpenOffice, et inspirent la création de licences libres telles que

5 Il est intéressant de constater que, dans ce texte, le modèle de la cathédrale est également (et peut-être surtout) assimilé au projet Emacs, que dirigeait et dirige toujours Richard Stallman. Il serait donc plus juste de dire que le texte *La cathédrale et le bazar* compare Linux et Emacs plutôt que le logiciel libre et le logiciel propriétaire.

Creative Commons ou Art Libre. De nombreuses autres initiatives, telles que les Archives ouvertes ou encore l'idée d'une « Open Source Science », mobilisent également les métaphores de la liberté et de l'ouverture des « sources » pour susciter l'adhésion des usagers et participants. Finalement, notons que la notion de code source s'introduit progressivement dans la culture populaire, comme en fait foi la publication du roman de William Gibson (2008b)(2008b) - *Spook Country* - dont la traduction française est intitulée « Code source » (Gibson, 2008a)(Gibson, 2008a), et d'un film du réalisateur Duncan Jones portant également ce même titre (Jones, 2011).

1.1.2 Le code source comme forme expressive

> Free software is an issue of free speech when we're moving more of our lives on to computers.
>
> Peter Brown, *Free Software Foundation*[6]

Nous avons déjà abordé la manière dont l'accès au code source est articulé comme un enjeu moral par les militants associés à la *Free Software Foundation*, en soulignant brièvement la manière dont le code source est progressivement devenu associé, dans le discours des acteurs, à une forme d'expression propre à être protégée par le droit à la liberté d'expression (ou aux États-Unis : par le premier amendement). Ces discours sur le code (source) comme forme d'expression ont été particulièrement analysés par l'anthropologue Gabriella Coleman. Déjà en 2003, dans une ethnographie de la communauté de militants du logiciel libre, Coleman propose de considérer la programmation des logiciels libres comme une forme spécifique d'exercice culturel de la liberté d'expression (Coleman, 2003). C'est cependant dans un article récent intitulé « Code is speech » (Coleman, 2009) que cette auteure analyse plus profondément ce rapprochement, dans les discours des acteurs, entre code[7] et forme d'expression (« speech[8] »). Selon cette auteure, le premier texte ayant circulé largement et

---

6 Tel que cité par Byfield (2006).

7 Comme nous l'expliquerons plus loin dans ce chapitre, il existe une certaine ambiguïté dans ce texte de Coleman entre les notions de « code » et de « code source », ambiguïté qui, bien que n'étant pas explicitée, reflète probablement une ambiguïté similaire dans le discours des acteurs. Nous aborderons cette ambiguïté tout au long de la thèse, mais en particulier à la section 1.2 du présent chapitre *(Qu'est-ce que le code source ? Premières définitions)* de même qu'au chapitre 4.

8 Nous avions initialement traduit le terme « speech » par « langage », mais nous avons finalement opté pour le terme « forme d'expression », étant donné l'importance de la référence à la « liberté d'expression » – freedom of speech – dans le discours des acteurs.

liant code source et forme d'expression est le « Freedom of Speech in Software » (Salin, 1991) où l'auteur caractérisait les programmes informatiques comme des écrits (« writings ») pour soutenir qu'ils étaient par conséquent impropres à être brevetés, mais appropriés pour être soumis aux protections liées à la liberté d'expression (au même titre que les autres écrits). Ce discours sur le code comme expression aurait ensuite été renforcé par une série de procès qui auraient plus ou moins confirmé cette adéquation. Coleman identifie un second moment clé de l'articulation juridique entre code source et forme d'expression dans le procès Daniel Bernstein contre le ministère de la Justice des États-Unis (*Bernstein v. U.S. Department of Justice*), concernant le droit de publier et d'exporter le code source d'un logiciel de cryptographie. Aux États-Unis, en effet, certaines formes de cryptographie sont régulées par les lois sur le trafic d'armes. Il n'est donc pas possible pour le développeur d'un logiciel de cryptographie de publier et en particulier d'exporter le code source de son logiciel à moins de s'enregistrer comme un trafiquant d'armes. Bernstein conteste cette réglementation en 1995 en soutenant que celle-ci viole le premier amendement de la constitution étasunienne concernant la liberté d'expression (*free speech*). En 1999, le juge donne raison à Bernstein en concluant que les régulations gouvernementales sur les logiciels cryptographiques violent effectivement le droit à la liberté d'expression.

Selon Coleman (2009), ce procès sert de base à bon nombre d'arguments des hackers concernant le droit de bricoler le code, qui est dès lors perçu comme un droit assimilable à la liberté d'expression. La lutte pour faire reconnaître le code source comme une forme d'expression s'amplifie en particulier autour de l'acte DMCA[9], qui vise, entre autres, à interdire la production et la circulation de dispositifs techniques qui pourraient mettre en péril les mesures techniques de protection des droits d'auteur (*Digital Rights Management*−DRM). En particulier, quelques lignes de code source qui iraient dans ce sens pourraient devenir illégales, ce qui est perçu par les militants du logiciel libre comme une violation de la liberté d'expression. L'une des interventions les plus marquantes défendant la nature expressive du

9 DMCA : *Digital Millennium Copyright Act*. Le DMCA est une loi étasunienne adoptée en 1998, dont le but est de lutter contre les violations du droit d'auteur à l'ère du numérique. Son équivalent français est la loi relative au droit d'auteur et aux droits voisins dans la société de l'information, la DADVSI.

code source est un *Amicus curiae*[10] d'un groupe d'informaticiens et de hackers, parmi lesquels Richard Stallman, où le code informatique est décrit comme un texte devant être considéré comme une forme de langage[11]. Le document insiste également sur le fait qu'il n'y a pas de différences notables entre les langages informatiques de haut niveau, exprimés dans le code source, et des langues naturelles comme l'allemand et le français.

> This court can find no meaningful difference between computer language, particularly high-level languages as defined above, and German or French. All participate in a complex system of understood meanings within specific communities [...]. The expression of ideas, commands, objectives and other contents of the source program are merely translated into machine-readable code[12].

Toutefois, si Coleman a sans doute raison de situer à l'année 1991 la premier document cherchant à articuler juridiquement code source et liberté d'expression, nous pouvons remonter beaucoup plus loin pour constater cette représentation de la programmation comme activité expressive. Coleman note en effet que l'idée de la programmation comme une forme d'écriture et d'expression prend également de l'ampleur à la fin des années 1960 à travers les publications du professeur Donald Knuth sur l'art de programmer (Knuth, 1968). Par exemple, dans une conférence présentée en 1974, Knuth propose une réflexion sur la transition des pratiques de la programmation informatique vers une discipline qui aurait des fondements scientifiques (la science informatique). Si Knuth appelle à une telle « scientifisation » de la programmation informatique, il considère toutefois la nécessité de continuer à l'appréhender comme une forme d'art, où le programmeur peut expérimenter un style, une manière de s'exprimer : « The important thing is that you really like the style you are using; it should be the best way you prefer to express yourself » (Knuth, 1974, p. 670).

Finalement, plusieurs ouvrages récents réalisés autant par des informaticiens que par des chercheur-es en sciences sociales ont abordé cette dimension expressive, voire esthétique, de

---

10 Selon le lexique juridique du site web *Juritravail.com*, l'expression *amicus curiae* désigne « la personnalité que la juridiction civile peut entendre sans formalités dans le but de rechercher des éléments propres à faciliter son information ».
<http://www.juritravail.com/lexique/Amicuscuriae.html> (consulté 15 novembre 2012).

11 <http://cryptome.org/mpaa-v-2600-bac.htm> (consulté le 17 février 2011)

12 <https://w2.eff.org/IP/Video/MPAA_DVD_cases/?f=20010126_ny_progacad_amicus.html> (consulté le 17 février 2012)

la pratique d'écriture du code source. Dans *Hackers and Painters,* Graham(2004) compare la programmation informatique à la peinture. Le livre *Beautiful Code* (Oram et Wilson, 2007) regroupe pour sa part une trentaine de textes rédigés par des informaticiens autour du thème de la « beauté du code », en présentant des cas de code « beau » et en soutenant l'analogie entre le code source et un essai de type littéraire. Également, dans sa thèse de doctorat, Auray (2000) souligne quant à lui la façon dont les concepteurs de Linux comparent leur production au Kevala, un poème épique finlandais. Finalement, citant Knuth, Krysa et Sedek (2008) considèrent pour leur part que le code source peut avoir des propriétés esthétiques et que par conséquent, l'idée même de code source dépasse le domaine de la programmation et du logiciel :

> The idea of source code, and indeed the open source model, extends beyond programming and software. For instance, Knuth points to creative aspects of programming alongside technical, scientific, or economic aspects, and says that a program "can be an aesthetic experience much like composing poetry or music". Source code can be considered to have aesthetic properties; it can be displayed and viewed. It can be seen not only as a recipe for an artwork that is on public display but as the artwork itself – as an expressive artistic form that can be curated and exhibited or otherwise circulated (Krysia et Grzesiek, 2008, p. 239).

Certains auteurs et militants du logiciel libre s'appuient sur cette dimension expressive du code source et de la programmation informatique pour justifier la politique du logiciel libre. C'est le cas d'Eben Moglen, professeur de droit à l'Université Columbia et associé de très près à la Fondation pour le logiciel libre. Pour Moglen (1999), la représentation de la technologie qui imprègne les systèmes de droits d'auteur ne tiendrait compte que de la dimension fonctionnelle des logiciels et exclurait sa dimension *expressive*. Notion importante en science informatique, l'expressivité fait référence à la capacité d'un langage de programmation d'exprimer textuellement des idées abstraites et ainsi de faciliter le travail en commun. Olivier Blondeau (2004; 2005) mobilise également cette notion d'expressivité en soutenant que le militantisme du code consiste à donner accès à la dimension expressive du code, à sa constitutivité esthétique, éthique et finalement politique; ou, pour employer les termes de Simondon (1958), à la prise de conscience de l'humanité des fonctionnements internes : « Ne sommes-nous pas aujourd'hui avec les hackers et le logiciel libre dans cette utopie

simondonienne réconciliant technique et culture dans une perspective d'émancipation ? » (Blondeau, 2005).

Les différentes références que nous avons présentées dans cette section insistent toutes sur cette dimension « expressive » du code source, que nous souhaitons mettre de l'avant dans ce chapitre. Pour beaucoup de militants du logiciel libre – ou militants du code (Couture et Proulx, 2008) – l'accès au code source ne doit pas seulement être garanti parce qu'il favorise un modèle de développement technique supérieur, mais surtout parce qu'il s'agit d'une question de liberté d'expression. Dans un certain sens, si l'on compare le code source à un poème, restreindre l'accès au code source équivaudrait à interdire la lecture du poème. Suivant cette analogie, on voit bien comment l'accès au code source apparaît pour les acteurs comme un enjeu de liberté d'expression qui va au-delà de la simple promotion d'un nouveau modèle de développement de logiciel (comme le soutiennent les militants de l' « open source », du moins ceux de la première heure). Plutôt que de simplement analyser les pratiques de collaborations *autour* du code source, le discours des acteurs sur le code source comme forme expressive devrait au contraire amener le chercheur à s'intéresser également à l'objet lui-même – le code source – et à ce qui, dans ses propriétés ou dans ses configurations, facilite la collaboration.

1.1.3 « Le code, c'est la loi », ou la performativité du code

> « This is code in its modern sense. It regulates in the ways I've begun to describe. The code of Net95, for example, regulated to disable centralized control; code that encrypts regulates to protect privacy » (Lessig, 2006, p. 72)

Un autre axe de notre problématique s'inscrit dans une préoccupation concernant la manière dont certaines normes ou valeurs peuvent être inscrites dans le design des technologies et des infrastructures technologiques. En ce qui concerne le code informatique, cet aspect est notablement développé par Lawrence Lessig dans les différentes éditions de son ouvrage *Code and Other Laws of Cyberspace*. Dans cet ouvrage, Lessig (2006) propose de considérer les technologies de l'information à la manière de codes juridiques qui prescrivent ou limitent certains comportements. L'auteur affirme que les comportements humains sont régulés par quatre types de contraintes, soit la loi, le marché, les normes sociales et finalement le monde

« physique », Lessig faisant référence au dernier aspect par la métaphore de l'architecture et de ses codes techniques. Par exemple, l'une des manières de réguler la consommation du tabac serait de réglementer la quantité de nicotine dans les cigarettes, c'est-à-dire de réguler le code des cigarettes. La réflexion de Lessig est cependant surtout concernée par Internet, qu'il appelait en 1999 le cyberespace, dont l'architecture est formée de code informatique.

Cette perspective trouve écho dans plusieurs travaux contemporains qui s'attardent à cette capacité des artefact à agir, ou à « faire de la politique ». Andrew Feenberg (2004), utilise ainsi un vocabulaire très similaire à celui de Lessig dans un questionnement sur l'autorité législatrice des code techniques :

> Les codes techniques qui façonnent notre vie reflètent des intérêts sociaux particuliers auxquels nous avons délégué le pouvoir de décider dans quel lieu et de quelle manière nous vivons, quel genre de nourriture nous absorbons, comment nous communiquons, nous nous divertissons, nous nous soignons, etc. L'autorité législatrice de la technique augmente à mesure qu'elle se fait de plus en plus envahissante. Mais si la technique est si puissante, pourquoi n'est-elle pas soumise aux mêmes normes démocratiques que celles que nous imposons aux autres institutions politiques ? (Feenberg, 2004, p. 109)

Cette préoccupation pour le pouvoir « législatif » du code trouve un écho particulier dans plusieurs réflexions théoriques contemporaines sur la capacité d'action, voire le rôle des artefacts. Elle est par exemple centrale dans l'essai de Langdon Winner (1980; 2002), « Les artefacts font-ils de la politique », aujourd'hui bien connu, et où l'auteur soutient que les artefacts techniques, en eux-mêmes, font de la politique en ce sens qu'ils peuvent incorporer différentes formes de pouvoir et d'autorité. Allant dans un sens similaire, Serge Proulx mobilise pour sa part le terme de « prescription d'usage » pour décrire la manière dont « à travers la forme que lui donne le concepteur, [le design de l'objet technique] induit des contraintes et une pragmatique de son usage virtuel » (Proulx, 2005a, p. 313). Proulx donne plusieurs exemples de ces « prescriptions d'usage » : le design horizontal et hétérarchique d'une architecture de réseau qui induit un système de communication décentralisé; la fonction « correction » d'un logiciel de traitement de texte qui induit systématiquement des biais erronés dans la conception de mots non répertoriés dans son dictionnaire; le niveau d'ouverture de l'architecture matérielle d'un ordinateur qui oblige ou non à acheter différents périphériques (Proulx, 2005a, p. 313). Dans un numéro de *Sociologie du Travail*, Jérôme

Denis approfondit une réflexion similaire en parlant d'artefacts prescriptifs qui « inscrivent les règles au cœur de l'exécution » (Denis, 2007). L'auteur donne l'exemple de la pratique informatique du « copier/coller », commune à un grand nombre de logiciels informatiques, qui agit souvent de façon prescriptive en guidant l'usager dans la façon dont il positionne les éléments de son document : « le copier/coller d'une image témoin, déjà habillée « comme il faut », permet l'élaboration de nouveaux éléments qui s'alignent sur la structure générale » (Denis, 2007). Notons finalement que les plus récents de ces travaux se tournent de plus en plus vers la notion de *performativité* pour saisir cette capacité d'action des artefacts. Cet aspect sera développé plus profondément dans le chapitre 2.

Un aspect qui nous intéresse ici pour notre problématique est de saisir le lien entre la dimension expressive du code et sa dimension performative, ou plus précisément, entre l'activité d'écrire du code et sa performativité. Malgré le pouvoir que Lessig (2006) attribue à cette pratique du code dans la régulation des comportements, il est étonnant de constater que cette pratique elle-même ne semble pas analysée dans sa démarche. Ainsi, nulle part Lessig ne semble analyser la façon dont ces « code writers » discutent ou jugent des choix concernant l'architecture du code. C'est un peu comme si un sociologue du droit nous parlait du pouvoir régulateur des textes législatifs, sans jamais nous donner un seul exemple de ces textes ou de la manière dont ceux-ci sont construits ou négociés. Comment analyser l'articulation entre, d'une part, le travail d'écriture et de fabrication du code source et, d'autre part, sa performativité ?

1.1.4 Rapports d'autorité et travail invisible dans la fabrication du code source

> One gains authority not by a structure that says, you are the sovereign, but by a community that comes to recognize who can write code that works (Lessig, 1998, p. 114).

Un dernier axe de notre problématique, par ailleurs étroitement lié au précédent, concerne les rapports d'autorité et le travail invisible dans la fabrication du code source. Dans la perspective où le code informatique régule effectivement les comportements humains, Lessig insiste sur le fait que ceux et celles qui écrivent le code informatique, les « code writers », deviennent de plus en plus des législateurs (« lawmakers »). Ce sont eux qui écrivent le code qui détermine les configurations par défaut de l'Internet, le fait que la vie privée sera protégée

ou non, le degré d'anonymat qui sera préservé, etc. Les écrivains du code, pour Lessig, sont ceux qui fixent la nature de l'Internet (« They are the ones who set its nature ») (Lessig, 2006, p. 72). L'objectif de Lessig, dans ce livre, est d'insister sur la nécessité d'une plus grande régulation, de la part des pouvoirs publics, du travail d'écriture du code : « Comment le code régule, qui sont ceux qui écrivent le code, et qui contrôlent ceux qui écrivent le code. Voilà des questions que toute pratique de justice doit poser à l'ère du cyberespace[13] » (Lessig, 2006, p. 79). Adoptant une perspective similaire, Brousseau et Moatty (Brousseau et Moatty, 2003) mobilisent également la catégorie de la « régulation par le code », en la contrastant avec les formes de régulations étatiques basées sur la violence légitime. Comme pour Lessig, la régulation par le code renvoie pour ces auteurs au fait que « l'écriture de lignes de code informatique commandant l'exécution de procédures revient à imposer aux utilisateurs des normes d'usages » (Brousseau et Moatty, 2003, p. 8).

Dans un article plus ancien, Lessig (1998) aborde toutefois plus spécifiquement la question des logiciels libres et à code source ouvert. Ainsi, pour Lessig, le mouvement du logiciel libre rejetterait l'autorité du pouvoir formel, celui des rois, des présidents et du vote. Ainsi, dans une telle dynamique, les acteurs acquièrent une autorité, non parce qu'une structure les déclare souverains, mais parce qu'une communauté reconnaît qu'ils sont en mesure d'écrire du code fonctionnel. Dans une telle perspective, l'autorité ne repose pas sur des règles formelles, mais trouve au contraire sa source dans le code fonctionnel et le consensus approximatif (« This is not to say the Net rejects authority. Indeed [it] identifies the source of authority – "rough consensus and running code" – and between the two, I suggest, it is the second that is more important ») (Lessig, 1998, p. 114). Selon Lessig, donc, chacun peut accéder au code source de l'Internet et l'améliorer :

> Without anyone's permission, one can improve it. Without going to work for any single company, one can work on it. One can improve it without getting the permission of some governmental bureaucrat. One can improve it without getting the authority of an election. [...] Anyone can propose a direction for the code, but proposals are not in English; proposals are in code. If there is a bug you think should be fixed in a particular way, you fix it, and offer the code that fixes it (Lessig, 1998, p. 113)

13 « How the code regulates, who the code writers are, and who controls the code writers, these are questions on which any practice of justice must focus in the age of cyberspace » (Lessig, 2006, p. 79).

Si nous rejoignons Lessig sur certains aspects de son analyse du pouvoir du code et de la nécessité, pour une pratique de justice, de questionner ceux qui écrivent le code, nous sommes par contre assez sceptiques quant à l'idéalisme exprimé dans cet article par rapport au modèle du logiciel libre. En particulier, la manière dont Lessig considère le « running code » comme la source fondamentale de l'autorité laisse invisible l'épaisseur des mécanismes d'autorité qui participent à la fabrication effective du « running code ».

L'un des traits symptomatiques de cette mise en invisibilité des mécanismes d'autorité concerne la faible visibilité, voire la faible présence des femmes dans le monde du logiciel libre. Un exemple de cette invisibilité est le sondage en ligne réalisé en 2002 dans le cadre de l'enquête européenne FLOSS, qui a révélé une proportion de 1,1 % de femmes parmi un échantillon de 2 784 « développeur-es de logiciels libres » (Ghosh, Robles, et Glott, 2002). Ce chiffre de 1,1 % de femmes dans le monde du logiciel libre soulève des interrogations importantes quant à l'affirmation de Lessig (voire même certains discours sur les logiciels libres) selon laquelle tout le monde peut proposer une direction au code (« Anyone can propose a direction for the code ») (Lessig, 1998, p. 114). Si tout le monde peut proposer une direction au code, comment expliquer alors que les femmes ne le font pas ?

Cependant, l'étude de Ghosh *et al.* (2002) est elle-même problématique puisqu'elle prend le « running code » comme source de l'autorité, en laissant invisibles les formes de participation subordonnées à l'autorité de ceux reconnus comme auteurs du « running code ». Dit autrement, la faible proportion de femmes mise de l'avant par Ghosh *et al.* (2002) pourrait être davantage analysée comme une omission méthodologique dans la prise en compte de certaines formes de participation, plutôt qu'à l'absence de cette participation elle-même. Il est en effet intéressant de savoir que la première étape de l'enquête de Ghosh *et al.* (2002) a consisté en une invitation sur différents forums de discussion électroniques, en ciblant spécifiquement et explicitement les « développeur-es ». Ce premier échantillon, qui réunissait 2 784 personnes, a ensuite été réduit à environ 500 personnes, en retenant que les répondant-es dont le nom figurait effectivement comme auteur-e dans le code source d'un logiciel. L'enquête de Ghosh *et al.* définissait donc la catégorie « développeur-es de logiciels libres » comme contributeur-e, ou auteur-e, d'une partie du code source. Cette définition est problématique, car elle ne permet pas de saisir un bon nombre de contributeurs et de contributrices au développement du logiciel libre, qui n'écrivent pas de code source et qui ne

sont donc pas reconnus comme auteurs du code source. Comme le note Haralanova (2010) dans son mémoire de maîtrise, les femmes sont souvent engagées dans différents aspects du développement de logiciels, même si elles n'écrivent pas de code source.

Bien que cette faible présence des femmes dans le monde du logiciel libre est sans doute l'une des expressions les plus significatives d'une hiérarchisation du travail et de l'autorité en lien avec le code source (et en particulier dans le monde du logiciel libre), notre intérêt dans cette thèse se portera davantage sur les rapports d'autorité, et la manière dont la participation de certaines catégories d'acteurs est privilégiée ou non, en particulier à travers le design du code source.

1.2 Qu'est-ce que le code source ? Premières définitions

Dans la section précédente, nous avons abordé plusieurs enjeux concernant le code source, sans toutefois définir précisément cette notion. Cette deuxième partie du chapitre fera ressortir différentes définitions du code source, depuis ses origines jusqu'à certaines formalisations plus contemporaines. Ce parcours nous amène à poser l'hypothèse que la formalisation de la notion de code source est récente et encore relativement instable. Nous terminons cette partie en proposant une définition « pragmatique » de la notion de code source qui nous permettra de cerner davantage notre objet d'étude. Sur le plan méthodologique, notons que la recherche historique mentionnée dans ce texte n'est évidemment pas extensive. Celle-ci a surtout pour objectif de mieux situer notre problématique.

1.2.1 Code source, code machine, compilation

Il est difficile de retracer un historique de la notion « code source », car celle-ci, bien qu'utilisée au moins depuis les années 1960 dans les publications des informaticiens, ne semble faire l'objet d'une formalisation qu'assez tardivement dans l'histoire de l'informatique

et est souvent nommée « programme source », voire seulement « source »[14]. Dans les articles scientifiques, la première mention de la notion de code source (en anglais) que nous avons trouvée est dans un article publié en 1965 dans un journal informatique nommé *Bit* (Jensen, 1965) et portant sur le compilateur Algol. L'article en question ne fait qu'une seule fois mention du terme anglophone « source code », en introduction. Dans le reste du texte, les termes « source text » et « source program » semblent utilisés dans le même sens. Il s'agit de générer un code machine (« machine code ») à partir d'un code source (« source code »), à l'aide d'un compilateur :

> The paper describes the method used in the GIER compiler for **generating machine code** from an error free Reverse Polish representation of the **source code** (Jensen, 1965, p. 235)[15].

Un autre article publié sur le même sujet deux ans plus tôt dans le même journal (Naur, 1963) ne mentionne pas quant à lui le terme « code source », bien que le mot « source » reste utilisé dans les termes « source text » ou « source program », qui semblent être utilisés de manière synonyme. On retrouve une situation similaire dans l'article de Backus et al. (1957) qui décrit le fonctionnement du premier compilateur et qui mentionne à plusieurs reprises les termes « code » et « source », sans toutefois utiliser le double terme « code source ». Les auteurs parlent par exemple de « source language », « source program », « source statement », « source program » :

> If the **object program** is to work correctly, every symbolic index register must be so governed that it will have the appropriate contents at every instant that it is being used. It is the **source program**, of course, which determines what these appropriate contents must be (Backus et al., 1957, p. 193).

Dans une analyse sommaire des dictionnaires de l'informatique disponibles à Télécom ParisTech, nous avons également constaté que le terme « code source » (ou « source code », en anglais) n'apparaît pas de façon régulière dans tous les ouvrages. Ainsi, l'édition de 1992

---

14 Richard Stallman, par exemple, n'utilise pas la notion de « code source » (ni celle de « logiciel libre » d'ailleurs) dans son annonce initiale du projet GNU, en 1983 : <http://www.gnu.org/gnu/initial-announcement.html> (consulté le 24 novembre 2011). Par contre, on retrouve la notion de « code source » dans le manifeste GNU, publié deux ans plus tard, puis dans la définition du logiciel libre : <http://www.gnu.org/gnu/manifesto.en.html> (consulté le 19 février 2011).

15 Nous soulignons.

du *Macmillan Encyclopedia of Computer* (Bitter, 1992) ne contient pas d'entrée pour le terme « source code », bien que l'entrée « compiler » mentionne les termes de « source language » et « source program ». La situation est similaire pour le *Dictionnaire de l'informatique* qui inclut une entrée « Source » définie comme « langage différent du langage machine, servant au programmeur pour écrire un programme, lui-même appelé "programme source" » (Ginguay, 1987, p. 237). On retrouve cependant une entrée « code source » dans le *Dictionnaire de l'informatique* de Illingworth qui définit ce terme comme une « Forme d'un programme qui est entré dans un compilateur (compiler) ou dans un traducteur (translator) pour être converti en code objet équivalent » (Illingworth, 1991, p. 527). Le *Dictionary of Computer and Internet Terms* contient également le terme « source code » définit comme « a programming language designed for use by human beings, as opposed to object code, which is used internally in the computer. A compiler translates source code into object code » (Downing, Covington, et Covington, 2000, p. 450). Les autres ouvrages que nous avons analysés – le *Dictionnaire des technologies de l'information et de la communication* (Astier, 2001), le *Dictionary of Multimedia and Internet Applications* (Botto, 1999), le *Dictionnaire anglais-français d'informatique* (Ginguay, 2005) et le *Vocabulaire international de l'informatique* (Association française de normalisation, 1986) – ne contiennent pas le terme « code source », ni le terme « source ».

Nous pouvons voir dans ces différentes définitions que les notions synonymes de « code source », « programme source » ou simplement « source » sont souvent opposées aux notions de « code objet », « programme objet » ou encore « code machine » ou « langage machine ». Certains auteurs, tels Illingworth (1991, p. 527) font également référence à la notion de *compilateur* qui transforme le code source en code objet, ou qui agit comme traducteur entre ces deux formes que peut prendre le code informatique. Pour mieux comprendre cette opération de traduction ou de transformation, rappelons qu'un ordinateur ne peut comprendre que le code binaire, c'est-à-dire un code formé à la base de 0 et 1. Pour manipuler un ordinateur, il faut donc ultimement fournir des instructions dans un tel code binaire. Chacun de ces codes binaires représente une instruction qui sera exécutée directement par la machine. Programmer à l'aide du code binaire est évidemment assez ardu, et assez tôt dans l'histoire de l'informatique, des langages informatiques dits de « haut niveau » ont été inventés, ceux-ci étant plus lisibles par les humains et pouvant par la suite être automatiquement traduits en

langage dit de « bas niveau » et éventuellement dans le langage machine, binaire, et directement « compréhensible par l'ordinateur ». Le compilateur est le dispositif qui procède à cette traduction entre un code élaboré dans un langage de « haut niveau » – le code source, davantage lisible par l'humain – et un code écrit dans un langage de « bas niveau » – le code objet, plus directement traitable par l'ordinateur.

Un dernier aspect à noter est que si l'on peut constater une relation entre code source et langage, en revanche, à aucun moment dans les ouvrages que nous venons de survoler, le code source n'est directement décrit comme un texte ou un écrit.

1.2.2 Le code source, une notion instable et en construction

Il semble que ce ne soit qu'à partir du début du troisième millénaire qu'apparaissent des efforts pour définir beaucoup plus formellement le concept de code source informatique. À partir de 2001, un cycle de conférences parrainé par l'Institute of Electrical and Electronics Engineers (IEEE), intitulé *Source Code Analysis and Manipulation* (SCAM)[16], s'est donné pour mission de rassembler des chercheurs qui s'intéressent à l'analyse et à la manipulation du code source des systèmes informatiques. Selon le site web, cette initiative s'appuie sur le constat que plusieurs des aspects de l'ingénierie logicielle sont couverts par différentes recherches, mais que le code source reste négligé dans ces études, alors qu'il constitue la « seule description précise du fonctionnement d'un logiciel[17] » :

> While much attention in the wider software engineering community is properly directed towards other aspects of systems development and evolution, such as specification, design and requirements engineering, it is the source code that contains the only precise description of the behaviour of the system. The analysis and manipulation of source code thus remains a pressing concern.

Le site web de ces conférences définit le code source comme « n'importe quelle description exécutable d'un système informatique » (« any fully executable description of a software system [18] », en mentionnant explicitement que cette définition très large du code source permet d'inclure le code machine, des langages de « très haut niveau » (« very high level languages ») et des représentations graphiques exécutables des systèmes informatiques.

---

16 <http://www.ieee-scam.org/> (consulté le 24 novembre 2011).

17 <http://cist.buct.edu.cn/zheng/scam/2010/indexse10.html> (consulté le 24 novembre 2011).

Les documents produits par les militants du logiciel libre sont également intéressants à analyser pour comprendre cette généralisation de la définition du code source. Ainsi, la première version de la Licence publique générale GNU (GPL), publiée en 1989, définissait le code source de cette manière :

> Source code for a work means the preferred form of the work for making modifications to it. For an executable file, complete source code means all the source code for all modules it contains [...][19].

Cette définition est intéressante, car elle dissocie la notion de code source de celui de « logiciel » en faisant plutôt de cet artefact la forme privilégiée pour faire des modifications. Les versions suivantes de la GPL précisent la définition du code source pour le mettre en lien avec le concept de « travail exécutable » :

> The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable[20].

La version 3 de la licence GPL[21] conserve essentiellement la même définition, mais ajoute une nouvelle notion de « Corresponding source code » que nous n'aborderons pas ici.

C'est cependant dans la définition du code source donnée dans l'*Amici Curiae* des militants du logiciel libre, que nous avons mentionné plus tôt, que l'on retrouve la définition la plus générale du code source :

> A more proper definition of source code may be simply that it is that subset of human expression which computers can interpret and execute[22].

---

18 La définition complète du code source selon ce site : « For the purpose of clarity 'source code' is taken to mean any fully executable description of a software system. It is therefore so-construed as to include machine code, very high level languages and executable graphical representations of systems. The term 'analysis' is taken to mean any automated or semi automated procedure which takes source code and yields insight into its meaning. The term 'manipulation' is taken to mean any automated or semi-automated procedure which takes and returns source code ». <http://cist.buct.edu.cn/zheng/scam/2010/indexse10.html> (consulté le 24 novembre 2011).

19 <http://www.gnu.org/licenses/gpl-1.0.html> (consulté le 24 novembre 2011).

20 <http://www.gnu.org/licenses/gpl-2.0.html> (consulté le 24 novembre 2011).

21 <http://www.gnu.org/licenses/gpl-3.0.html> (consulté le 20 février 2012).

22 <http://cryptome.org/mpaa-v-2600-bac.htm> (consulté le 17 février 2011).

Nous pouvons voir que ces dernières définitions sont très générales. D'une part, le code source est défini comme la « forme privilégiée du travail pour réaliser des modifications ». D'autre part, le code source est également défini comme « un sous-ensemble des expressions humaines que les ordinateurs peuvent interpréter et exécuter ». Si ces définitions nous amènent à considérer la difficulté à saisir notre objet, elles montrent en revanche que la pertinence d'analyser le code source va bien au-delà des sciences informatiques. En particulier, la dernière définition du code source comme sous-ensemble de l'expression humaine nous semble la plus significative d'un point de vue anthropologique ou communicationnel.

Pour terminer, mentionnons deux autres définitions contemporaines du « code source » qui abordent le caractère écrit de cet artefact. D'abord, celle de Wikipédia :

> Le code source (ou les sources voire le source) est un ensemble d'instructions écrites dans un langage de programmation informatique de haut niveau, compréhensible par un être humain entraîné, permettant d'obtenir un programme pour un ordinateur[23].

Ensuite, celle donnée dans le chapitre « Code Source » du livre *Software Studies* :

> Source code (usually referred to as simply « source » or « code ») is the uncompiled, non-executable code of a computer program stored in source files. It is a set of human readable computer commands written in higher level programming languages (Krysia et Grzesiek, 2008)

Cette brève revue historique des écrits nous amène à insister sur plusieurs aspects. D'abord, comme nous l'avons mentionné plus tôt, la formalisation de la notion de code source est assez récente. Ensuite, du fait que cette notion est en cours de formalisation, on remarque une instabilité du terme, qui s'exprime à travers différentes contradictions. Par exemple, alors que (Krysia et Grzesiek, 2008) définissent le code source comme étant le « code non-exécutable », la conférence SCAM, à laquelle nous avons fait référence plus tôt, définit sur son site web le code source comme une « description exécutable » d'un système logiciel. Si cette contradiction peut être seulement apparente, en revanche, d'autres questions importantes surgissent : qu'en est-il de la documentation, et en particulier, des commentaires intégrés dans les mêmes fichiers que le code source ? Font-ils partie du code source ? Le code source doit-

23 <http://fr.wikipedia.org/w/index.php?title=Code_source&oldid=52775534> (consulté le 6 juillet 2010).

il être « lisible humainement » pour être considéré en tant que tel ? Comme Wikipédia le souligne, quel est le niveau d'entraînement nécessaire pour pouvoir comprendre le code source ? S'agit-il toujours d'un texte, d'un écrit ?

Les recherches que nous avons réalisées à ce sujet nous amènent à poser l'hypothèse que la définition du code source est relativement floue et en voie d'être formalisée. Cette hypothèse trouve explicitement écho dans l'*Amici Curiae* de plusieurs informaticiens, que nous avons présenté plus tôt :

> Indeed, even « source code » itself is a relative term. The notion of what it is changes rapidly as developments in computer technology occur, and it is particularly important for the Court to understand that point as it considers even how to define the term, let alone how, if at all, it may be restricted[24].

D'une certaine manière, c'est cette relativité du terme « code source » et les changements rapides dans sa définition qui permettent le mieux de faire correspondre notre objet d'étude à « la nature complexe et mouvante des objets des sciences sociales » (Latzko-Toth, 2010, p. 33).

1.3 Le code source, un objet d'étude négligé. Recension des écrits

L'exposé de nos principaux axes de problématique a déjà fait ressortir plusieurs travaux qui ont abordé la question du code, ou du code source. Dans cette troisième partie du chapitre, nous proposons une recension plus systématique des écrits en sciences humaines et sociales concernant le code et le code source. Cette recension des écrits se fera sur trois axes. Le premier concerne les travaux qui mobilisent la catégorie du « code », voire du code informatique, de façon surtout métaphorique ou alors dans le cadre d'une réflexion plutôt sociopolitique que pragmatique. Le deuxième axe sera consacré à la place du code source dans les études sur les logiciels libres qui abordent l'artefact code source, sans toutefois en faire leur principal objet d'étude. Finalement, nous présenterons des travaux plus récents qui posent beaucoup plus explicitement le code, et dans certains cas, le code source, comme objets d'études, et s'identifiant parfois au courant émergent des *Code Studies*. L'un des constats que nous faisons de ces différents travaux est que le terme même de *code source* n'est que très rarement problématisé, les auteurs utilisant la plupart du temps uniquement le

24 <http://cryptome.org/mpaa-v-2600-bac.htm> (consulté le 17 février 2011).

terme « code » pour décrire ce qui nous semble être du « code source »; dans d'autres cas, les termes de « code » et de « code source » sont utilisés de manière synonymes, sans que le rapport entre ces deux termes soit explicité dans leur travail.

Cette recension des écrits ne prétend pas faire un portrait exhaustif de l'ensemble des études qui peuvent, ou non, avoir abordé la question du code. En effet, de nombreuses études ont été réalisées sur de multiples aspects de l'Internet, et il en est de même concernant les logiciels libres. Notre objectif n'est donc pas de faire une revue exhaustive des écrits, mais plutôt de positionner notre étude face à certains ensembles de travaux.

1.3.1 Le code informatique dans les études en communication

Avant d'aborder la place du code informatique dans les études en communication, écartons d'emblée les travaux qui font usage des notions de code, codage et décodage, dans un sens conceptuel qui n'a que très peu à voir avec le code source informatique, dans un sens concret et pragmatique. Écartons par exemple le texte « Codage/décodage » de Stuart Hall (1994)qui, bien que pertinent pour l'étude des communications, ne renvoie pas au code informatique, au sens empirique. Notre étude ne concerne pas non plus le concept de « code technique », mis de l'avant par Feenberg pour faire référence aux « aspects de ces régimes techniques qu'il faut interpréter comme le reflet direct de valeurs sociales significatives » (Feenberg, 2004). Si ce concept est éclairant sur le plan théorique, son utilisation par Feenberg, dans un sens philosophique et général, n'a a priori pas directement à voir avec le code source, encore une fois au sens concret et pragmatique du terme.

Plusieurs travaux abordent le concept de code dans un sens plus ambigu en ce qui concerne notre recherche, en faisant allusion parfois au code informatique, mais parfois aussi au code d'une manière plus floue ou générale. L'ouvrage *Code* de Lessig (2000; 2006) est assez significatif de cette instabilité empirique. La catégorie du « code » y est utilisée pour faire référence parfois de manière générale au design des technologies, parfois encore aux standards de l'Internet et à d'autres moments, aux lignes d'instructions qui font fonctionner l'ordinateur. Ceux que Lessig appelle les « code writers » sont parfois désignés comme des acteurs individuels, par ailleurs souvent des dirigeants : « the sort of people who governed the Internet Engineering Task Force » (Lessig, 2006, p. 71) – et à d'autres moments comme des institutions telles le FBI (p. 35).

Plusieurs réflexions contemporaines qui s'attardent aux aspects sociaux et politiques de l'Internet abordent également la question du code informatique. Certains auteurs tels que Aigrain (2005) analysent par exemple les transformations économiques et techniques actuelles en tant qu'elles favorisent l'émergence et l'importance grandissante d'« industries du code » qui tirent profit de la propriété du code et de la connaissance codifiée (brevets, protocoles, standards techniques, logiciels, etc.). Nous avons nous-même participé à un projet de recherche qui, s'inspirant des réflexions de Aigrain et Lessig, s'attardaient aux pratiques et valeurs de ceux que nous avons nommés des *militants du code* (Proulx, 2006; Proulx, Couture, et Rueff, 2008) Cette étude analysait les pratiques de petits collectifs qui militent pour une appropriation démocratique des technologies de l'information, c'est-à-dire des technologies basées sur le code. Si cette étude trouve sa richesse dans l'analyse qu'elle fait des pratiques et des valeurs de ces militants, la catégorie du code y occupe une place qui demeure surtout de l'ordre de la métaphore, plutôt que d'un véritable objet empirique[25]. Bien que ces travaux abordent des aspects qui nous semblent cruciaux dans l'étude des communications, ils diffèrent de l'approche de notre thèse par le fait qu'ils ne traitent pas directement du code informatique, en tant qu'artefact, mais plutôt des enjeux politiques autour du code (ou derrière le code). En outre, comme met en garde Kittler (2008), il est important de ne pas succomber à la tendance actuelle à sur-utiliser la notion de code pour caractériser des situations ou des phénomènes qui n'ont rien à voir, de fait, avec le code informatique, voire même le concept de code tout court.

Une fois ces distinctions établies, quelle place occupe le code informatique, et plus spécifiquement, le code source, en tant qu'objet d'étude empirique, dans les études en communication ? Bien que de plus en plus de recherches s'attardent à comprendre les différents aspects de l'Internet et des technologies de l'information, il semble que le « logiciel » et plus spécifiquement le « code source » des logiciels, dans leur épaisseur matérielle, demeurent des objets encore remarquablement négligés (Fuller, 2008). À titre

25 D'ailleurs, les publications réalisées dans le cadre de cet ouvrage ne reprennent pas toutes la catégorie du code. Ainsi, l'étude a donné lieu à un ouvrage dont le titre faisait simplement mention de l'action communautaire à l'ère du numérique (Proulx, Couture, et Rueff, 2008) , tandis qu'une autre publication faisait quant à elle mention des technoactivistes (Proulx, 2007a). C'est d'ailleurs en partie comme prolongement de cette étude que nous avons entrepris cette présente réflexion sur le code source.

d'exemple, une brève analyse que nous avions réalisée en début de thèse des courriels publiés sur la liste de discussion de l'*Association of Internet Researchers*[26] (Association des chercheurs de l'Internet) faisait ressortir que sur 3 380 courriels envoyés sur cette liste entre le 5 mars 2007 et le 31 janvier 2008, 287 courriels contenaient l'expression « software », 139 contenaient l'expression « code » tandis qu'une seule contenait l'expression « source code ». Notons toutefois que la plupart de ces études ont plutôt tendance à négliger l'étude du processus pragmatique d'écriture du code source, en utilisant par exemple le terme « code » comme une métaphore, sans toutefois analyser ce que constitue exactement cet artefact.

Cette situation se retrouve également dans l'ouvrage récent de Suchman (2007), *Human-Machine Reconfigurations*, auquel nous avons déjà fait référence et qui constitue un élément théorique de notre réflexion (voir chapitre 2). Cet ouvrage, développé en s'inspirant de plusieurs domaines d'études tels que l'étude des interactions humain-machine (Human Computer Interactions), les études sur les sciences et les technologies, les études féministes et les études médiatiques, a pour objectif de reconceptualiser l'interaction sur des bases différentes de celles qui conçoivent l'humain et la machine comme autonomes et a priori séparés, interagissant par l'intermédiaire d'une interface « humain-machine ». Ainsi, plutôt que de parler de « conversation à l'interface », Suchman suggère de mobiliser les métaphores de la lecture et de l'écriture pour décrire nos relations avec les artefacts computationnels. Étonnamment, Suchman n'aborde que très superficiellement les pratiques de programmation et la plupart du temps, surtout pour les critiquer et sans vraiment les analyser finement. En particulier, le code source informatique, un « artefact computationnel[27] » pourtant indispensable à la conception des technologies de l'information, est quant à lui complètement ignoré dans l'analyse de Suchman. Cette absence est également étonnante lorsque l'on considère que le code source constitue bien souvent un « texte », au sens pragmatique du terme (plutôt qu'au sens métaphorique, comme dans les *Cultural Studies*) pour lequel l'interaction passe nécessairement par certaines formes d'activités d'écriture et de lecture.

---

26 <http://listserv.aoir.org/listinfo.cgi/air-l-aoir.org> (consulté le 20 février 2012).

27 Nous pouvons cependant nous questionner sur le caractère « computationnel » du code source. En fait, l'une des particularités du code source est que, s'il constitue une description exécutoire de la computation, en revanche, il n'est peut-être pas lui-même « computationnel ». Le code source pourrait peut-être alors être décrit comme un artefact « précomputationnel ».

On peut cependant noter deux types de travaux qui ont plus spécifiquement analysé le code source de manière empirique : les travaux sur les logiciels libres et les travaux plus récents que l'on pourrait regrouper sous l'appellation des *Code Studies*.

1.3.2 La place du code source dans les études sur le logiciel libre

De nombreux travaux ont été réalisés dans les dernières années sur les projets, communautés et mouvements du logiciel libre. La place du code source en tant que tel est cependant ambiguë dans ces travaux. Plusieurs de ces études se sont surtout attardées aux *discours* des acteurs, sans analyser le code source lui-même comme artefact. Ainsi, dans sa thèse de doctorat, Nicolas Auray (2000) fait référence à plusieurs endroits au code source et remarque la façon dont les concepteurs de Linux comparent leur production à un poème épique finlandais. Il est toutefois remarquable de ne trouver aucun extrait de code source dans les 586 pages de sa thèse. Dans une perspective différente, Blondeau (2004) propose que le mouvement du logiciel libre constitue une lutte pour donner accès à la constitutivité esthétique, éthique et politique du code, sans pour autant analyser cette « constitutivité » pragmatique du code. Les travaux de Coleman sont également significatifs de cette approche. L'auteure se concentre surtout sur la manière dont les acteurs et actrices considèrent le code comme une forme de parole, sans toutefois approfondir particulièrement la notion de « code source », en tout cas, sans porter son analyse sur l'artefact lui-même. L'étude plus récente de Kelty (2008) concernant la signification culturelle du mouvement du logiciel libre nous semble aller dans le même sens. Dans cette superbe ethnographie des acteurs des logiciels libres, autant en Amérique du Nord qu'en Inde, l'auteur documente les rencontres des acteurs et leurs discours politiques sur le logiciel libre, mais également sur la culture libre, comme les *Creative Commons*. Le « code source » occupe une place importante dans l'index, et l'auteur présente par exemple l'histoire de différents systèmes de gestion de contenu. Toutefois, tout comme dans la thèse d'Auray soutenue huit ans plus tôt, on ne retrouve encore une fois aucun extrait, ou aucune figure, montrant à quoi pourrait ressembler le code source.

D'autres études ont abordé plus spécifiquement les *pratiques* de collaboration des acteurs des logiciels libres, mais en accordant une place plutôt périphérique à l'artefact code source. Par exemple, Demazière, Horn et Zune (2008; 2006; 2007a) ont étudié le cas du logiciel libre SPIP (qui est également un cas dans notre étude). Ils insistent brièvement sur les relations de

pouvoir et d'autorité concernant les droits en écriture du code source. Les auteurs notent ainsi les différenciations statutaires qui s'opèrent entre les contributeurs, différenciations qui participent à la régulation du projet dans son ensemble (Demazière, Horn, et Zune, 2006). Cependant, encore une fois, le code source en tant que tel y est complètement absent.

S'inscrivant dans la perspective des études sur les sciences et les technologies, Yuwei Lin en appelle à analyser le rôle des artefacts et des différents mondes sociaux dans l'univers du logiciel libre et open source (Lin, 2004; Lin, 2005). L'une des particularités de son étude est justement de montrer des extraits de code source (Lin, 2004, p. 121,244,247). Dans le chapitre 7 en particulier, l'auteure montre la manière dont la production de la connaissance, et en particulier du code source, dans les projets de logiciels libres est ancrée localement dans des problèmes particuliers et concrets à résoudre. Lin indique alors souhaiter voir les études sur le logiciel libre adopter un « tournant situé » (« situated turn ») en insistant sur le caractère situé des connaissances en jeu dans les processus de résolution des problèmes.

Notons finalement l'étude FLOSS, réalisée en 2002, qui s'appuie notamment sur l'analyse des auteurs inscrits dans le code source pour construire son échantillon de répondants (Ghosh, Robles, et Glott, 2002). Une section du rapport de recherche (22 pages) est consacrée à l'analyse du code source de certains logiciels. Les auteurs cherchent notamment à extraire des informations sur les auteurs, la grosseur ou l'intégrité du code source, de même que sur les dépendances entre les projets. Les auteurs séparent le code source en différentes composantes telles que la documentation, écrite en langage naturel comme l'anglais, ou les en-têtes (headers), qui performent des relations entre différents fichiers du code source. Si l'analyse est intéressante, le concept de code source lui-même n'est par contre pas défini et tout se passe plutôt comme si ce concept fait déjà l'objet d'une compréhension commune.

1.3.3 *Software Studies* et *Code Studies*

Une dernière série de travaux se sont attachés beaucoup plus attentivement à l'étude du code informatique et pourraient être regroupés autour du courant des *Critical Code Studies* (CCS). Ce dernier terme a été mis de l'avant dans le titre d'un article publié par Mark Marino (2006) et a également été l'intitulé d'une série de conférences en ligne qui réunissaient une centaine de chercheurs en février et mars 2010. Dans son texte le plus récent sur le sujet, Marino

(2010) situe le courant des CCS dans la lignée des *Software Studies*[28] et des *Platform Studies*[29], mais en s'intéressant plus spécifiquement au code. Il définit les CCS comme « une application de l'herméneutique à l'interprétation de la signification extra-fonctionnelle du code source informatique ». Il propose d'analyser le code comme un texte, au sens de la sémiotique comme un système de signes (Marino, 2006), mais également au sens des *Cultural Studies*, en tant qu'il se situe dans un contexte matériel et historique (Marino, 2010). Par exemple, Marino propose de considérer le programme *Hello World*, comme le point d'entrée de l'alphabétisme numérique. Mentionnons finalement que les *Critical Code Studies* seraient surtout concernés par l'interprétation, en laissant les aspects pragmatiques à d'autres courants des *Code Studies*. De fait, comme nous allons l'expliquer plus loin, notre approche se distinguera de celle de Marino par le fait qu'elle sera beaucoup moins interprétative et plus pragmatique.

D'autres efforts récents captent cependant particulièrement notre attention par la manière dont ils posent le code comme premier objet de l'analyse. C'est en particulier le cas d'une édition de la revue *TeamEthno-Online* ayant pour titre « Ethnographies of Code : Computer Programs as the Lived Work of Computer Programming » (Rooksby et Martin, 2006). L'objectif de ce numéro était de rassembler des recherches qui tentent d'analyser les pratiques de design informatique ou les fondements des sciences informatiques. Les textes réunis proviennent de disciplines diverses telles que l'informatique, la sociologie, la psychologie et la philosophie, mais se situent au confluent de la tradition anglophone du HCI (*Human-Machine Interactions*). Le texte intitulé « Knowledge and reasoning about code in a large code base », rédigé par les éditeurs du numéro (Martin et Rooksby, 2006), présente par exemple une ethnographie des conversations verbales dans une entreprise de développement de logiciels. Dans leur analyse, les auteurs montrent comment la compréhension du code ne se fait pas seulement par sa lecture, mais implique aussi un processus d'essais-erreurs et de conversations plus ou moins formelles qui amène les programmeurs à analyser plus finement une partie du code. Un autre texte de la série, « Designing a program. Programming the design » (Kristoffersen, 2006) s'inspire explicitement des approches ethnographiques et

28 <http://lab.softwarestudies.com/2007/05/about-software-studies-ucsd.html> (consulté le 20 décembre 2011).

29 <http://platformstudies.com> (consulté le 20 décembre 2011).

ethnométhodologiques des études des CSCW (*Computer Supported Cooperative Work*) et HCI (*Human-Computer Interactions*). L'article soutient ainsi que la programmation doit être appréhendée comme un élément constitutif du design des technologies de l'information et non pas comme une simple expression codifiée d'un design qui aurait été élaboré extérieurement : « Programming cannot (or should not) be seen as a "result" of design, which ideally and productively could be mapped from "a" design » (Kristoffersen, 2006, p. 10).

Un autre effort similaire, qui ne s'associe cependant pas explicitement au courant des *Code Studies*, est l'étude d'Adrian Mackenzie (2005; 2006) sur le code informatique[30]. S'inscrivant surtout dans les *Cultural Studies* mais en faisant notamment référence à la théorie de l'acteur-réseau, Mackenzie propose de considérer le code informatique comme un objet-culture (*culture-object*) qui est façonné, articulé, senti et modifié par la pratique et les différentes situations (Mackenzie, 2006, p. 7). Il propose cette définition formelle du logiciel et du code : « A set of permutable distributions of agency between people, machines and contemporary symbolic environments carried as code. Code itself is structured as a distribution of agency » (Mackenzie, 2006, p. 19). Dans son analyse, Mackenzie note explicitement l'ambiguïté de sa définition du « code ». Il distingue ainsi deux modes d'existence du logiciel, soit le code source « what programmers read and write » (Mackenzie, 2005, p. 90) et le logiciel lui-même : « for instance, the Linux kernel – is principally 'binary code' produced by compiling the 'source code' » (Mackenzie, 2005, p. 90). Cependant, dans son essai, il est souvent difficile de savoir si son objet d'analyse est le code comme « logiciel lui-même » ou le code comme « code source ».

Parmi les travaux ayant abordé la question du code informatique dans la perspective des études sur les sciences et les technologies (STS), mentionnons également une série de table-rondes que nous avons nous-même organisée lors d'un congrès du *Society for the Social Study of Sciences* (Couture et Cohn, 2009). Ces sessions cherchaient à articuler les perspectives appréhendant le code – ou le logiciel – comme un langage, ou un texte, et le « coding » comme une activité ou un engagement. La session souhaitait réunir des présentations qui répondraient à des questions telles que : comment l'artefact code peut-il être

30 Mackenzie (2005) propose également une analyse de la performativité du code informatique, sur laquelle nous reviendrons au chapitre suivant.

conceptualisé en tenant compte qu'il est souvent produit par une multiplicité d'auteurs – « is often multiply authored » (Couture et Cohn, 2009) – et qu'il existe dans différentes localisations ? Comment rendre compte des différents modes d'existence du code, comme logiciel et comme code source ? Quelles sont les formes de travail avec les ordinateurs qui sont considérées comme du « codage » et lesquelles ne le sont pas ? Les présentations, provenant de différents horizons, ont abordé par exemple, les relations entre le sens et l'action dans le code (Buswell, 2009) la comparaison entre les codes des télégraphes et ceux du tricotage (Haring, 2009) l'usage de codes de couleurs dans les premières œuvres d'art numérique (Kane, 2009); le code comme forme de conversation dans certaines méthodologies de développement de logiciels (Cohn, 2009); les codes utilisés dans le protocole RSS de syndication des blogues (Passoth, 2009); la valeur de l'« ouverture » dans le développement de logiciels (Knouf, 2009); ou encore l'imaginaire et le design du code ailleurs dans le monde, comme au Vietnam (Nguyen, 2009) ou en Chine (Lindtner, 2009)

Dans une tout autre perspective, mentionnons finalement le travail remarquable de Clarisse Herrenschmidt (2007). Dans son livre *Les trois écritures, langue, nombre, code*, l'auteure propose de considérer l'écriture informatique et réticulaire comme le troisième système d'écriture, après la langue et la monnaie. L'objectif de cette spécialiste des écritures anciennes est de « construire les linéaments d'une sémiologie historique et anthropologique des écritures du vieil Orient, de l'Europe et de l'Occident [...] » (Herrenschmidt, 2007, p. 190). Elle analyse tout d'abord l'émergence de l'alphabet – la première écriture – en s'attardant à l'invention de l'écriture de la civilisation élamite et en analysant de façon très détaillée la construction des caractères de l'alphabet. Dans une deuxième partie, l'auteure s'attarde à l'écriture des nombres – la deuxième écriture – depuis la création de la monnaie frappée dans l'antiquité jusqu'à l'usage des nombres dans l'économie et les statistiques modernes. Présentée par l'auteure comme une « ouverture » dans ses travaux, la troisième partie est quant à elle consacrée à l'écriture informatique – la troisième écriture – dont elle situe l'origine dans l'article de Turing sur l'intelligence artificielle (Turing, 1936; Cité par Herrenschmidt, 2007, p. 432). L'écriture informatique n'est pas seulement liée au traitement de texte (ce qu'elle analyse par ailleurs), mais est également articulée à l'écriture d'un programme informatique. Bien que cet ouvrage

est très éloigné de la perspective que nous développons dans cette thèse, la mention de celui-ci permet de situer notre travail dans un cadre anthropologique et historique plus large[31].

1.4 Objectifs de recherche et question centrale

Cette dernière partie du chapitre présente de façon formelle notre objet d'étude de même que nos objectifs et questions de recherche. Nous terminons ce chapitre en explicitant ce que nous considérons être la contribution de notre thèse à l'étude sociologique de la communication.

1.4.1 Objet d'étude spécifique : le code source du web 2.0

Avant de présenter nos objectifs de recherche, il est important de préciser ici que nous n'aborderons pas ici tous les types de code source, mais plus précisément le code source de logiciels qui sont utilisés comme infrastructure de ce qu'il est convenu aujourd'hui d'appeler le « web 2.0 ». Comme nous le décrirons plus en détail, notre étude s'attardera à deux projets qui ont en commun d'être écrits dans le langage PHP, un langage informatique aujourd'hui utilisé par de nombreux sites web notables, tels Facebook et Wikipédia, qui forment ce qu'il est aujourd'hui courant d'appeler le « Web 2.0 » ou le « web participatif ». PHP est un acronyme récursif qui signifie **P**HP: **H**ypertext **P**reprocessor[32]. Il s'agit d'un langage de programmation orienté vers la conception de sites web dits « dynamiques » ou « participatifs ».

Certaines recherches statistiques considèrent que PHP est aujourd'hui le quatrième langage de programmation le plus populaire pour la conception de logiciels informatiques, tous usages confondus[33]. D'une certaine manière, nous pourrions qualifier PHP comme la colle, ou le ciment, permettant d'assembler plus ou moins rigoureusement les objets techniques formant le web 2.0[34]. Ce choix d'étudier des projets écrits en PHP nous permet notamment d'inscrire

---

31 Pour une recension de cet ouvrage, voir Weill (2007).

32 Un acronyme récursif est un acronyme qui fait appel à la récursivité ou à l'auto-référence, où par exemple, la première lettre de l'acronyme désigne l'acronyme lui-même. L'usage d'acronymes récursifs est courant dans le monde du logiciel libre, par exemple, dans le cas du projet GNU, mentionné plus tôt, qui signifie *GNU is not Unix*.

33 Voir le site <langpop.com> (consulté le 20 février 2012) qui propose une « comparaison normalisée » de l'usage des différents langages de programmation à partir, notamment, du résultat de certaines recherches dans Yahoo, Google, ou dans les listes de discussions.

notre recherche comme une contribution aux études concernées par le « web social » (Proulx, 2007b; Millerand, Proulx, et Rueff, 2009).

```
<?php
    $nomsAutorises = array('Albert', 'Bertrand');
    $nomEnCours    = 'Eve';
?>

<?php if (in_array($nomEnCours, $nomsAutorises)) :
?>

    <!-- code HTML -->
    <p>Bonjour <?php echo $nomEnCours ?> !</p>

<?php else : ?>

    <!-- code HTML -->
    <p>Vous n'êtes pas un utilisateur autorisé !
</p>

<?php endif ?>
```

Figure 1.1 : Le langage de programmation PHP.
Exemple tiré de la page Wikipédia PHP :
<http://fr.wikipedia.org/w/index.php?title=PHP&oldid=75469398> (consulté le 20 février 2012).

1.4.2 Objectifs et questions de recherche

Notre approche de recherche s'appuiera en grande partie sur le développement théorique récent de Suchman sur les reconfigurations humain-machine, auquel nous avons fait référence plus tôt, et que nous présenterons plus en détail dans le prochain chapitre. Comme indiqué précédemment, l'ouvrage de Suchman (2007), *Humain-Machine Reconfigurations*, est une édition largement enrichie de son ouvrage marquant, *Plans and Situated Actions*, mais avec quelques chapitres supplémentaires dont l'objectif principal est de repenser les configurations de plus en plus intimes et intriquées entre l'humain et la machine[35]. Les

34 Perl, un langage de programmation plus ancien et ayant inspiré PHP, était d'ailleurs considéré comme le « duct tape » de l'Internet, étant donné sa prégnance et son caractère peu élégant. Voir <http://oreillynet.com/pub/a/oreilly/perl/news/importance_0498.html> (consulté le 20 février 2012).

35 « To rethink the intricate, and increasingly intimate, configurations of the human and the machine » (Suchman, 2007, p. 1).

notions d'interfaces et de reconfigurations sont particulièrement importantes, autant dans cet ouvrage de Suchman que dans notre présente étude. L'interaction humain-machine, dans la perspective développée par Suchman, ne consiste donc pas tant en une conversation entre humains et machines par l'intermédiaire d'une interface, mais est plutôt appréhendée comme la coproduction d'un monde sociomatériel partagé. Cependant, l'interface elle-même, en tant que frontière entre humains et machines, n'est pas donnée a priori, mais constitue plutôt le site de reconfigurations intimes, continuelles et de plus en plus étroitement intriquées des relations entre humains et machines. Comme toute frontière, l'interface est par ailleurs porteuse de conséquences politiques. Comme Suchman l'indique, les pratiques de démarcation des frontières constituent une mise en œuvre de la différence et sont donc nécessairement politiques.

Notre premier objectif, qui peut sembler assez trivial, mais qui est néanmoins important, est de produire une recherche qui prend spécifiquement pour objet le « code source ». Ainsi, bien que certains aspects de la présente thèse recoupent des aspects qui ont été explorés dans d'autres études, la particularité de notre thèse est d'orienter la perspective sur la compréhension de ce qui constitue exactement le « code source ». Un second objectif est d'ordre plus politique et consiste à s'attarder aux dimensions proprement politiques du code source et à la manière dont cet artefact participe à l'organisation des collectifs médiatisés par ordinateur. Finalement, notre troisième objectif est d'insister plus particulièrement sur le caractère écrit du code source. Bien que le code source ne constitue pas dans tous les cas un artefact, il reste néanmoins que dans les projets étudiés, il prend très souvent la forme d'un texte. Comment appréhender les similitudes entre le code source et d'autres artefacts écrits, en particulier les wikis ? Quelles sont les conséquences de l'organisation du code source sur l'organisation des collectifs humains médiatisés par ordinateur ?

Cette recherche s'inscrit dans le sillon des travaux de Suchman et a pour objectif d'analyser le rôle du code source dans l'évolution et les reconfigurations humain-machine, en nous attardant plus spécifiquement au processus collectif de fabrication du code source. Notre recherche s'attarde plus spécifiquement à l'étude de la place du code source dans *SPIP* et *symfony*, deux logiciels libres et à code source ouvert, utilisés dans la fabrication de sites web interactifs et de médias sociaux. Ce choix permet de situer notre recherche en continuité des

recherches contemporaines qui abordent les différents aspects de l'Internet et du « web social » (Millerand, Proulx, et Rueff, 2009).

Suivant ces différentes perspectives, que nous explorons plus tard, notre recherche vise à répondre aux questions spécifiques suivantes :

- Comment les acteurs définissent-ils le code source et en particulier, dans quelle mesure cet artefact peut-il être appréhendé comme un écrit ?

- Comment la fabrication et la stabilisation du code source s'articulent-elles à sa force « législative » ou performative ?

- De quelles manières certaines valeurs sont-elles inscrites dans la fabrication et le design du code source?

- De manière plus générale, comment le code source agit-il dans la reconfiguration d'Internet?

Ces différentes questions spécifiques de recherche peuvent être regroupées autour de notre question centrale de recherche, d'ordre théorique, et dont les termes sont davantage approfondis dans le chapitre suivant :

> *Qu'est-ce que le code source et comment cet artefact agit-il dans les reconfigurations d'Internet?*

1.4.3 Contribution à l'étude sociologique de la communication

Serge Proulx (2012), en introduction d'un ouvrage qui interroge les interrelations entre la sociologie de la communication et les études sur les sciences et les technologies (STS), soutient l'intérêt de suivre de près les études sur le code informatique. Pour Proulx, ces études participent du « projet d'ouvrir la boîte noire des technologies informatiques à partir d'une déconstruction des activités de programmation du code » (Proulx, 2012). Selon cet auteur, les analyses du code informatique trouvent leur pertinence dans la manière dont elles concernent le noyau des infrastructures technologiques avec lesquelles nous interagissons quotidiennement. Par le fait même, Proulx remarque que le choix d'objets de recherche tels que le code informatique permet d'établir des correspondances conceptuelles entre la sociologie de la communication médiatisée et les études STS, au point où il apparaît de plus

en plus difficile de distinguer les deux types d'approche. Pour Proulx, ces arrimages entre sociologie de la communication et STS, que permet le choix d'un objet d'étude tel que le code informatique, amèneraient « à penser la technique autrement, en prenant en compte les contraintes et possibilités de ses déterminations complexes, mais en nous éloignant des apories de la pensée déterministe » (Proulx, 2012).

Notre étude sur le code source permet de contribuer à la sociologie de la communication à travers plusieurs des aspects soulignés par Proulx. D'une part, l'étude du code source permet en quelque sorte d'ouvrir la boîte noire des technologies logicielles et d'appréhender les configurations complexes et changeantes qui se situent dans le « noyau » des technologies logicielles et des infrastructures technologiques. Parmi les autres travaux concernés par le code informatique et les logiciels libres, notre étude trouve cependant son originalité dans la manière dont elle prend explicitement le *code source* comme objet d'étude, tout en problématisant la définition de ce terme. Tel que mentionné dans ce chapitre, de nombreuses études ont abordé plus ou moins explicitement les logiciels et le code informatique, mais aucune étude, à ce jour, ne semble avoir explicitement problématisé la notion de *code source.* Or, cette notion spécifique, distincte de celle plus générale de code, prend une importance politique et épistémique[36] de plus en plus grande. Ainsi, si certains auteurs abordent effectivement ce qui nous semble être le code source, c'est souvent la notion de code *tout court*, voire celle de logiciel, qui est utilisée. Nous espérons ainsi que notre étude pourra contribuer à problématiser davantage la notion de code source, et que cette problématisation pourra servir de base pour des analyses subséquentes.

Sur le plan empirique, nous souhaitons dans cette thèse mettre de l'avant la manière dont le code source constitue non seulement le résultat, ou le produit de l'activité des programmeurs, mais également le site même de la collaboration entre les acteurs. D'une certaine manière, il s'agit de montrer que le code source, en particulier dans le cadre du « web 2.0 », n'est pas seulement le noyau des infrastructures technologiques, mais qu'il peut être appréhendé en lui-même comme une forme médiatique, typique du web 2.0. Comme nous l'aborderons en

---

36 Sur le plan épistémologique, nous avons par exemple noté que la notion de code source, qui n'était pas appréhendée dans les dictionnaires de l'informatique il y a une quinzaine d'années, fait désormais l'objet d'une page dédiée sur Wikipédia et que plusieurs conférences et articles scientifiques sont également désormais consacrés à cet objet.

profondeur, notre étude pourrait en effet se situer à l'horizon d'une anthropologie des formes d'écritures numériques et hautement distribuées, tels que les wikis ou même les « tweets » qui mélangent langage naturel et certains « codes » dictant la mise en page ou facilitant l'agrégation de contenu. Notre projet rejoint finalement celui de Simondon, centré sur une prise de conscience du sens des objets techniques. Il s'agit donc de mettre de l'avant la dimension expressive du code source, c'est-à-dire de montrer que l'étude du code source n'est pas seulement pertinente à analyser parce qu'elle concerne le noyau des infrastructures technologiques, mais également, et peut-être surtout, parce que le code source est appréhendé par les acteurs comme une forme d'expression qui doit être respectée pour elle-même.

CHAPITRE II

PERSPECTIVE THÉORIQUE : LE TRAVAIL D'ASSEMBLAGE HUMAIN-MACHINE COMME PROJET CULTUREL ET POLITIQUE

> The alternative perspective suggested here takes persons and machines as contingently stabilized through particular, more and less durable, arrangements whose reiteration and/or reconfigurations is the cultural and political project of design in which we are all continuously implicated (Suchman, 2007, p. 286).

Ce chapitre présente l'approche théorique qui guide nos analyses. Le fil rouge de notre développement théorique s'articule autour du concept de performativité et s'inscrit en continuité de la perspective développée dans l'ouvrage de Lucy Suchman, *Human-Machine Reconfigurations* (Suchman, 2007). La première partie tente de situer notre approche de recherche à partir de différents axes qui ont mobilisé notre réflexion. La deuxième partie approfondit certains concepts clés qui caractérisent l'orientation de notre regard. Finalement, la troisième partie aborde certains enjeux épistémologiques et moraux de l'approche choisie, que nous considérons comme étant située à l'horizon du pragmatisme. L'objectif de cette dernière partie est de tracer quelques pistes d'articulation de notre étude à des réflexions beaucoup plus larges et philosophiques sur la matérialité et les relations entre humains et machines. Notre objectif n'est pas tant ici de présenter un cadre théorique rigide servant à l'analyse des données, comme le serait par exemple une approche hypothético-déductive, mais plutôt de présenter un « paysage théorique » des différents travaux qui alimentent notre réflexion.

2.1 Situer notre approche de recherche

Nous présentons dans cette première partie du chapitre les différents axes de recherche au fondement de notre analyse. La première section, plus introductive, présente les travaux sur la sociologie des usages, dans lesquels nous enracinons notre étude. Les deux sections suivantes présentent respectivement la théorie de l'acteur-réseau, puis certains travaux, en

particulier dans une perspective féministe, auxquels Hess (2001) fait référence comme la deuxième génération des études STS. Cette dernière section présente par ailleurs plus en profondeur les travaux récents de Lucy Suchman (2007). Nous terminerons cette première partie par une présentation succincte de la pensée de Simondon (2001; 2007), qui forme en quelque sorte l'arrière-plan philosophique de notre réflexion.

2.1.1 De la sociologie des usages à l'étude des sciences et technologies

Notre étude est d'abord ancrée dans la sociologie francophone des usages des technologies de l'information. Cette tradition s'est constituée dans le paysage français des années 1980, à la suite notamment de l'introduction du Minitel qui a donné lieu à des formes d'usages très diversifiées. Ces études se sont établies entre autres en opposition à une vision trop déterministe n'accordant aucune liberté à l'usager dans l'appropriation des technologies. Les premières études sur les usages sont influencées par le travail précurseur de Michel de Certeau (1990) qui insiste notamment sur la créativité des « gens ordinaires » qui bricolent leur quotidien, parfois en opposition aux dictats des élites ou du système établi. Suivant cette approche, les premières études sur les usages des technologies de l'information mettent de l'avant la part de créativité des usagers dans leur appropriation des objets techniques. Elles insistent par exemple sur les différentes formes d'appropriation d'une même technologie, ou encore sur la manière dont les usagers détournent l'usage d'une technologie pour répondre à certains besoins (Breton et Proulx, 2002).

Les études sur les usages se sont progressivement intéressées à l'objet technique lui-même, de même qu'aux activités de conception. Au tournant du millénaire, Millerand (1998) puis Breton et Proulx (2002) font déjà référence aux travaux de la sociologie de l'innovation de Callon, Latour et Akrich (Callon et Latour, 1986; Akrich, Callon, et Latour, 2006) – aujourd'hui connue sous le terme de théorie de l'acteur-réseau – comme l'un des axes programmatiques dans l'étude des usages des TIC. Ces études sur les usages ont ensuite donné naissance à des modèles plus complexes prenant en compte plusieurs niveaux d'analyse incluant les activités de conception et la manière dont les objets techniques, en particulier à travers leurs interfaces, participent aux relations entre usagers et concepteurs et transportent des valeurs morales et politiques plus larges (Proulx, 2005b).

La prise en compte des activités de conception devient encore plus importante avec l'émergence, sur Internet, de pratiques décrites par certains chercheurs comme des « innovations par l'usage ». Ce terme, mis de l'avant par Eric Von Hippel (2005) puis revisité notamment par Cardon (2005) sous la notion d' « innovation ascendante », permet de saisir des formes d'innovation qui procèdent de l'usage au lieu d'être conçues dans des laboratoires. Von Hippel (2005) donne l'exemple des serres sur la planche à voile inventées par les surfeurs eux-mêmes pour améliorer leur pratique. Ces dynamiques d'innovation par l'usage sont particulièrement observées dans le cadre de la coopération dans le monde du logiciel libre. Contrairement aux formes d'innovation classiques, les innovations par l'usage ne visent pas à répondre à des clientèles cibles. Au contraire, dans ce type d'innovations, quelquefois à base coopérative comme dans le cas des logiciels libres ou de l'encyclopédie Wikipédia, les usagers apportent eux-mêmes des solutions pour répondre à leurs propres besoins et intérêts.

L'analyse des innovations par l'usage s'est progressivement déplacée vers l'analyse de médias sociaux, tels que Facebook, Twitter ou encore Wikipédia, communément appelés « web social » ou encore « web 2.0 ». Ces travaux mettent notamment de l'avant la figure de l'*usager contributeur*, qui ferait éclater les anciennes frontières séparant production et consommation ou encore, innovation et usage (Proulx, 2012; Proulx et al., 2011). Au centre de ces nouvelles formes d'usages, on retrouve la notion de la *contribution*, terme par ailleurs fortement mobilisé par les acteurs eux-mêmes. Cette notion, sur laquelle nous reviendrons plus loin dans ce chapitre, est aujourd'hui conceptualisée par les chercheurs tantôt comme une forme sociale (Proulx et al., 2011, p. 9), tantôt comme une transaction économique qui ne relèverait ni du don ni de la transaction marchande (Goldenberg, 2010; Licoppe, Proulx, et Cudicio, 2010).

Ces frontières ébranlées, voire éclatées, entre activités de conception et activités d'usage favorisent bien évidemment un rapprochement des études portant sur les usages des technologies et celles portant sur leur conception. C'est dans cette perspective que plusieurs chercheurs en communication s'intéressent aujourd'hui de plus en plus aux études sur les sciences et les technologies, souvent désignées par l'acronyme STS (*Science and Technologies Studies*). Serge Proulx, dans un article où il fait le point sur les interrelations entre ces deux domaines d'études, soutient que l'intérêt des sociologues de la communication pour les études STS s'explique principalement par la forte pénétration des technologies

numériques dans de nombreuses sphères de la communication humaine. Pour Proulx (2012, p. 17), « les perspectives et les choix d'objets de recherche de ces sociologues de la communication médiatisée apparaissent aujourd'hui très proches de ceux des sociologues s'identifiant aux STS », au point où il serait même aujourd'hui difficile de distinguer ces deux types d'approches. Pour notre part, notre démarche est surtout alimentée par les études STS, que nous décrivons dans la prochaine section.

2.1.2 Les études STS de première génération et la théorie de l'acteur-réseau

Le domaine des études STS n'est pas homogène et on y retrouve plusieurs courants parfois divergents. Nous souhaitons ici reprendre la distinction que fait Hess (2001) entre deux générations d'études STS. La première, marquée notamment par la théorie de la construction sociale des technologies et la théorie de l'acteur-réseau, est abordée dans cette section. La seconde génération, émergeant de la rencontre avec les études féministes et les études culturelles, sera discutée plus loin[37].

Selon Hess (2001), la première génération des études STS émerge au début des années 1980 avec pour ancêtre le courant de recherche dit de la sociologie de la connaissance scientifique (*Sociology of scientific knowledge* – SSK), encore appelé le « programme fort » (*Strong Program*). Ce courant de recherche se veut lui-même une réaction à la sociologie des sciences d'alors, associée aux travaux de Merton (1973). Bien que Merton proposait une sociologie des institutions scientifiques, son approche laissait largement de côté le travail quotidien des scientifiques. La science y était également appréhendée comme un processus fondamentalement rationnel et les critères universalistes tels que l'« évidence empirique » étaient perçus comme l'élément dominant de la clôture des controverses scientifiques (Hess, 2001). En réaction à cette approche, la sociologie de la connaissance scientifique met plutôt de l'avant l'idée d'une construction sociale des connaissances scientifiques. Les recherches plutôt ethnographiques réalisées dans cette perspective s'attachent notamment à montrer la manière dont la fabrication des sciences est étroitement et intimement articulée à des événements contingents et locaux, des processus de décisions et de négociations entre

37 Nous faisons également référence à l'article de Hess dans le chapitre suivant (sur la méthodologie de notre recherche) puisque l'article en question traite précisément de la place de l'ethnographie dans les études STS.

différents acteurs. Latour et Woolgar publient par exemple *La vie de laboratoire* (Latour et Woolgar 1979, 1988), une ethnographie qui décrit les conversations informelles, la manipulation des instruments scientifiques, les hésitations et l'insécurité des chercheurs, ainsi que le rôle du hasard dans l'élaboration des thèses et des faits scientifiques.

Plusieurs des chercheurs de cette première génération des études STS se sont progressivement tournés vers l'étude des technologies. L'une de ces approches, la théorie de la construction sociale des technologies (*Social Construction of Technologies* – SCOT), met l'accent sur le concept de flexibilité interprétative, soulignant que différents groupes peuvent construire des significations différentes d'une même technologie. Après un certain temps, cette flexibilité interprétative disparaît et le sens donné à la technologie se stabilise pour devenir dominant (Bijker, Hughes, et Pinch, 1987). Dans leurs premières recherches, les chercheurs de cette tradition se concentrent sur les premières étapes du développement technologique et cherchent à montrer comment une technologie se déplace de la flexibilité interprétative à la stabilité. Dans une période plus tardive, les chercheurs de l'approche SCOT abordent la question de la co-construction des groupes sociaux et de la technologie pour comprendre comment les usagers peuvent se regrouper et éventuellement modifier des technologies stables (Oudshoorn et Pinch, 2003)[38].

L'autre courant issu de la première génération des études STS et ayant abordé la question des technologies est la théorie de l'acteur-réseau (*Actor Network Theory* – ANT), appelée dans le contexte français *sociologie de l'innovation, sociologie de la traduction,* ou même *sociologie des sciences et techniques*. Cette approche, encore aujourd'hui dominante dans les études STS, a été notoirement développée dans le contexte français par Bruno Latour, Michel Callon et Madeleine Akrich (Akrich, Callon, et Latour, 2006), alors chercheurs au Centre de Sociologie de l'Innovation à Paris, et dans le monde anglophone par des chercheurs comme John Law (1991; 1999), Ann-Marie Mol (2002) et, de façon plus distante, Susan Leigh Star et Geof Bowker (Bowker et Star, 2000a; Star, 1995c). C'est ce courant qui retient notre attention ici.

---

38 Voir par exemple la thèse de Latzko-Toth (2010) qui aborde la co-construction du dispositif IRC.

La théorie de l'acteur-réseau n'est pas un modèle explicatif ou une théorie stable, mais pourrait plutôt être appréhendée comme une grammaire, ou un langage descriptif[39]. S'inspirant de différents courants tels que l'ethnométhodologie et la sémiotique, cette approche met de l'avant le principe d'une symétrie généralisée entre humains et non-humains. Cette symétrie permet de reconnaître la capacité d'action des objets et d'utiliser le même type d'explication pour tous les éléments qui composent un réseau hétérogène d'humains et de non-humains devenant progressivement solidifié. Latour (2006) décrit trois critères qui permettent de vérifier l'appartenance d'une recherche au corpus de la théorie de l'acteur-réseau :

> 1) La théorie de l'acteur-réseau accorde une attention particulière aux non-humains (objets naturels ou techniques): ils doivent être des « acteurs » et non seulement des objets possédant une certaine épaisseur symbolique;
>
> 2) Dans la théorie de l'acteur-réseau, il n'y a pas, au départ, de domaine spécifiquement social ou de domaine spécifiquement technique, ces catégories émergeant plutôt dans la stabilisation des réseaux entre acteurs humains et non-humains. Une explication qui prendrait comme point de départ une certaine essence ou une stabilité a priori du « social » ou de « la technique » ne se retrouverait donc pas au sein de la théorie de l'acteur-réseau[40];
>
> 3) La théorie de l'acteur-réseau vise à « rassembler le social ». En d'autres termes, cette approche s'intéresse à la manière dont les collectifs, ou les entités, se maintiennent et sont stabilisés, plutôt que d'insister sur leur dispersion ou leur déconstruction, ce qui serait – selon Latour (2006) – la perspective postmoderne.

---

39 Dans un texte mettant en scène un professeur et son étudiant souhaitant terminer sa thèse, Latour note à propos du cadre explicatif « que l'ANT est parfaitement inutile pour cela » et écrit : « N'essayez pas de basculer de la description à l'explication; contentez-vous de *prolonger* la description » (Latour, 2006, p. 213-214).

40 C'est probablement sur ce point que la théorie de l'acteur-réseau se distingue de l'approche de la construction sociale des technologies (SCOT) puisque dans cette dernière approche, « le social y est constamment maintenu dans un état de stabilité et sert à expliquer les modalités du changement technologique » (Latour, 2006, p. 21). En parlant de « construction sociale », cette approche postule de fait une certaine essence du domaine social ou du domaine technique, même si elle tente d'aborder les liens étroits qui s'établissent entre ces deux sphères.

La spécificité de la théorie de l'acteur-réseau est précisément de ne reconnaître a priori aucune structure, pas même les catégories aussi fondamentales que le social, la technique, l'humain et le non-humain : « une structure, c'est juste un réseau sur lequel on ne possède qu'une information très rudimentaire » (Latour, 2006, p. 224). Dans cette théorie, le monde est plutôt appréhendé comme un ensemble d'associations plus ou moins stabilisées ou, pour employer la terminologie de l'ANT, un ensemble de réseaux réunissant des acteurs humains et non-humains qui s'associent chacun selon leurs propres « intérêts ». De fait, le principal apport de la théorie de l'acteur-réseau est à notre sens le fait de réintégrer (voire simplement d'intégrer) la prise en compte des choses, artefacts et autres non-humains dans l'analyse sociologique.

L'application de la théorie de l'acteur-réseau dépasse aujourd'hui le cadre de l'étude des sciences et techniques au point où certains chercheurs s'identifiant à cette approche la présentent comme une sociologie générale, qui pourrait « refaire la sociologie » (Latour, 2006; Licoppe, 2008). Bruno Latour a par exemple réalisé des ouvrages qui s'attardent à la fabrication du droit (Latour, 2004). Antoine Hennion, chercheur au Centre de sociologie de l'innovation, travaille depuis longtemps à développer une sociologie de l'art proche de la théorie de l'acteur-réseau, qui prend notamment en compte la matérialité et la médiation dans l'élaboration du goût (Hennion, 2005; Hennion, 2007). Les travaux en anthropologie pragmatique de l'écriture, que nous aborderons plus loin, font également amplement référence aux travaux de Latour, Callon, Mol et autres théoriciens de la théorie de l'acteur-réseau, et s'intéressent notamment au rôle de la matérialité et au travail de stabilisation des artefacts écrits (Fraenkel, 2006; Denis et Pontille, 2010a; Denis et Pontille, 2010b).

2.1.3 Les études STS dites de « deuxième génération »

Hess (2001) situe l'émergence d'une « deuxième génération » d'études STS au milieu des années 1990, en particulier avec l'intégration à ce domaine d'études de chercheur-es d'origine étasunienne, et provenant de disciplines telles que l'anthropologie, les études féministes et les *Cultural Studies*. Pour Hess, ces études de seconde génération se démarquent notamment par la remise en question des principes d'impartialité et de symétrie qui étaient au centre des études de « première génération », telles celles associées au programme fort, à la théorie de l'acteur-réseau, ou à l'approche SCOT. Rappelons que le principe de symétrie consiste à

accorder le même type d'explication dans l'analyse de la construction des connaissances scientifiquement vraies qu'à celles scientifiquement fausses. Ce principe de symétrie a par la suite été généralisé dans la théorie de l'acteur-réseau pour appréhender de la même façon la capacité d'action des humains et des non-humains. Le principe d'impartialité, quant à lui, est le fait de ne pas distinguer a priori entre vérité et « fausseté » scientifique, entre rationalité et irrationalité; pour citer Latour, « il n'y a pas de groupe ni de niveau qu'il faille privilégier » (Latour, 2006, p. 44)

Les critiques du principe d'impartialité sont particulièrement importantes de la part de chercheures féministes qui reprochent aux auteurs STS de première génération de ne s'intéresser en fait qu'aux groupes sociaux déjà privilégiés. Judy Wajcman (2004) note par exemple que Bruno Latour, dans son livre *Aramis ou l'amour des techniques* (Latour, 1992a), ne s'intéresse qu'aux hommes occupant des positions bien visibles, en oubliant les femmes qui occupent quant à elles bien souvent des positions plus périphériques dans l'innovation technique. Elle note également que les approches STS dominantes, notamment la théorie de la construction sociale des sciences et la théorie de l'acteur-réseau, sont, ironiquement, réticentes à considérer le caractère politique de leurs propres méthodologies : « Practitioners act as if their own methodologies are not affected by the social context and have no politics » (Wajcman, 2004, p. 126).

En contraste avec les principes d'impartialité et de symétrie au cœur des études STS de première génération, Hess (2001) note que les études de deuxième génération mettent plutôt de l'avant l'idée d'une partialité[41], qui amènerait le chercheur, ou la chercheure, à privilégier certains groupes ou certaines « configurations » plutôt que d'autres : « For example, what alternatives are there to the current configuration of the production of content in the science and technology in a specific field of study ? » (Hess, 2001, p. 10). Ainsi, une étude

41 Le texte remarquable de Donna Haraway (1988) intitulé « Situated Knowledges: The Science Question in Feminism and the Privilege of Partial Perspective » constitue à notre avis un moment important de la conceptualisation de cette partialité. Dans ce texte, qui s'oppose notamment au « constructivisme radical » (*radical social constructionist program*, p. 577), Haraway soutient que l'objectivité féministe consiste à reconnaître le caractère situé et partiel de la connaissance qu'il devient par conséquent possible de soumettre à l'interrogation critique, au contraire d'une connaissance qui prétendrait ne venir de nulle part. Haraway soutient également que la position des subjugués (*subjugated*) constitue une position d'observation privilégiée (mais pas exclusive) dans la production de cette connaissance, car cette position serait plus sensible à l'effacement de la partialité de toute connaissance.

« impartiale » se limiterait à expliquer le succès ou l'échec d'une innovation technologique ou scientifique par différentes dimensions telles que les normes et les standards en vigueur à ce moment. La réinsertion d'une certaine partialité pourrait amener à conclure que ces normes et standards favorisent le statu quo ou privilégient des groupes déjà privilégiés. Dans cette perspective, soutient Hess, les études de seconde génération ont tendance à distinguer plus clairement l'impartialité méthodologique (ce que Hess appelle le *relativisme culturel*), nécessaire à l'analyse sociologique, d'une impartialité morale (ou *relativisme moral*) qui, sous prétexte de ne pas prendre position, privilégierait le statu quo.

De manière similaire, les études STS de seconde génération ont tendance à insister davantage sur la notion d'*asymétrie*. Pour illustrer l'asymétrie persistante entre croyances fausses et croyances vraies, Hess donne l'exemple de la croyance dans la sorcellerie et les phénomènes surnaturels, un exemple fréquemment mobilisé en sciences sociales. Une étude en sciences sociales analysera de la même manière la *croyance* dans les phénomènes naturels que dans les phénomènes surnaturels. Cependant, les phénomènes eux-mêmes n'auront généralement pas le même statut, selon qu'ils sont naturels ou surnaturels. Hess note d'ailleurs que Bloor lui-même avait reconnu cette asymétrie de haut niveau, sans toutefois l'approfondir :

> Likewise, Bloor recognizes a higher level asymmetry in the afterword to the second edition of *Knowledge and Social Imagery* (1991, p. 176). He argues that a sociological explanation of witchcraft – that is, as opposed to a supernatural explanation – « will logically imply that the witchcraft beliefs (taken at their face value) are false » (1991, p. 176). The logical asymmetry implicit in a sociological explanation of witchcraft is distinguished from the methodological symmetry of asking why members of a culture would choose the false belief – witchcraft is based on supernatural powers – over the true belief, witchcraft is not (p. 177). Bloor recognizes the problem of higher level asymmetry that arises from methodological symmetry, but his exploration of the implications of higher level asymmetry is limited (Hess, 2001, p. 6).

Le principe de symétrie généralisée entre humains et non-humains qui est au cœur de la théorie de l'acteur-réseau est également critiqué dans les études STS de seconde génération. Tout en reconnaissant la contribution du principe de symétrie, Lucy Suchman insiste ainsi pour réintégrer une certaine asymétrie, ou dissymétrie, dans les relations entre les humains et les choses :

> In particular, I would argue that we need a rearticulation of asymmetry, or more impartially perhaps, dissymmetry, that somehow retains the recognition of hybrids, cyborgs, and quasi-objects made visible through technoscience studies, while simultaneously recovering certain subject-object positionings – even orderings – among persons and artifacts and their consequences (Suchman, 2007, p. 269).

Ce sont les travaux plus récents de cette dernière auteure qui marquent le fil rouge de notre thèse (Suchman 2006; Suchman 2007; Suchman 2008). Cette auteure s'est d'abord fait connaître pour son ouvrage *Plans and Situated Actions: The Problem of Human-Machine Communication* (1987) qui étudie, à partir de la perspective d'une sociologie interactionniste, les usages d'une photocopieuse Xerox et en particulier de son interface interactive, alors à l'état de prototype. L'un des objectifs de cette étude est de comparer, à travers l'analyse empirique de ces interactions, les conceptions de la conversation alors prévalentes dans le champ de l'intelligence artificielle et basées sur les théories mathématiques de la communication, avec les théories sociologiques de la conversation qui émergeaient alors de l'analyse des conversations humaines en face à face. L'une des observations principales de l'étude est que la communication humaine ne se limite pas à la transmission de messages, telle que modélisée dans les théories mathématiques de la communication, mais repose également sur une forte prise en compte du contexte, ou de la situation (les mouvements gestuels ou du visage, le rapport à l'environnement immédiat et, plus généralement, le cours de l'action). L'étude montre par conséquent qu'il y a d'importantes différences entre l'humain et la machine au niveau de la communication. Ainsi, l'observation des interactions entre les humains et la photocopieuse montre plusieurs difficultés d'usages, difficultés qui s'expliquent par le fait que la machine n'a accès qu'à une partie infime des actions de l'usager pour communiquer. L'auteur conclut en définitive qu'il existe d'importantes et profondes asymétries entre les personnes et les machines qui, bien qu'obscurcies par le projet d' « interfaces interactives », refont surface dans la pratique sous la forme de nombreuses difficultés d'usages. Dans son ouvrage récent, elle reprend ce constat en insistant de nouveau sur cette asymétrie :

> With that said, my observation continues to be that although the language of interactivity and the dynamics of computational artifacts obscure enduring asymmetries of person and machine, people inevitably rediscover those differences in practice (Suchman, 2007, p. 13).

Dans ce dernier ouvrage, une réédition du précédent, Suchman cherche à articuler différents courants de recherche tels que la théorie de l'acteur-réseau, les interactions humain-machine (HCI), ainsi que les études féministes des sciences et des technologies (*Feminist STS*), notamment celles qui s'appuient sur la figure du cyborg mise de l'avant par Haraway (1985). Suchman reconnaît ainsi l'influence de la théorie de l'acteur-réseau et de son concept de symétrie pour mieux comprendre les relations entre humains et machines. Toutefois, elle note que d'insister sans discernement sur ce principe peut avoir des conséquences inattendues. Ainsi, si l'insistance sur le rôle des non-humains a l'avantage de favoriser une réintégration de la matérialité et de « la technique » dans les études en sciences sociales, celle-ci est en revanche problématique dans le milieu de l'ingénierie et de l'informatique où, loin d'être exclue, c'est « la technique » qui domine alors que « le social » est pour sa part laissé en marge des préoccupations dominantes (Suchman, 2007, p. 270). Proposer une symétrie entre humains et non-humains sans tenir compte de la façon dont ces catégories sont déjà distinctement privilégiées par des groupes sociaux particuliers, c'est courir le risque de reproduire ou d'aggraver des inégalités.

Pour terminer cette sous-section sur les études STS de seconde génération, mentionnons que selon Hess (2001), le caractère « socialement construit » des sciences et des technologies, en opposition à leur caractère purement rationnel, semble constituer une théorie acceptée plutôt qu'une préoccupation de recherche persistante. De manière générale, dans les études de seconde génération, la préoccupation se déplace vers les dimensions morales et politiques de cette « construction sociale », et vers les manières par lesquelles ces dimensions sont « inscrites » ou « encodées » au cœur même des artefacts scientifiques ou technologiques. Cette préoccupation est par exemple claire dans l'étude de Bowker et Star (2000a) sur différents systèmes de classification. où les auteurs démontrent la manière dont certaines catégories médicales participent d'une dynamique de visibilité et d'invisibilité dans l'ordonnancement des relations humaines. À cet effet, mentionnons finalement que la frontière entre les deux générations d'études STS est poreuse, certains des auteurs, comme Star et Bowker, se situant à la croisée des deux générations.

2.1.4 Les nouvelles figures de la performativité

Plusieurs études STS de première et de seconde génération se rejoignent aujourd'hui dans leur mobilisation du concept de performativité, influencées en cela par les études féministes et les *Cultural Studies* (Licoppe, 2008). Selon Licoppe, la perspective performative constituerait « un puissant outil de critique des postures essentialistes (qui considèrent que les entités ont des qualités préétablies) et représentationalistes (qui tiennent le langage comme capable de décrire un monde d'entités qui lui sont posées comme extérieures)[42] » (Licoppe, 2008, p. 294). Appliquée aux artefacts, la perspective performative permettrait également d'éviter une approche déterministe, en articulant l'effet des artefacts à un travail continuel et collectif de performation (Denis, 2006).

Le concept de performativité a été introduit par Austin (1991) en linguistique pragmatique pour analyser des situations où « dire c'est faire », c'est-à-dire pour expliquer des situations où le langage ne fait pas que constater une situation, mais constitue en lui-même un acte qui crée une nouvelle réalité. Par exemple, l'énoncé « je vous déclare mari et femme » n'a pas pour but de constater le mariage, mais plutôt de créer la réalité du mariage. La notion de performativité est aujourd'hui réutilisée dans de nombreux travaux. Pour Denis (2006), la reprise de la notion de performativité se démarque de l'usage initial d'Austin, par la manière dont elle abandonne la dimension proprement grammaticale de la performativité, pour plutôt s'attarder aux conventions, aux institutions, et plus généralement, aux « conditions de félicité » de la performativité. D'une certaine manière, ces travaux rejoignent la critique que fait Bourdieu à Austin, à savoir qu'un énoncé est performatif non pas uniquement parce qu'il obéit à certaines conventions formelles, voire grammaticales, mais parce que son énonciateur est institutionnellement autorisé à prononcer cet énoncé (Bourdieu, 1975). Cependant, les nouveaux travaux se distinguent également de Bourdieu par une prise en compte beaucoup plus large de la situation dans l'analyse de la performativité, en reliant, par exemple, la force performative à un travail de performation réitéré et souvent continuel, ou en considérant la dimension matérielle de la situation.

---

42 Karen Barad définit quant à elle le représentationalisme comme « l'idée selon laquelle les êtres existent comme individus, de façon antérieure à leurs représentations » (Barad, 2003, p. 804).

Parmi ces travaux, mentionnons ceux de Butler (1990; 1997), qui ont eu un impact significatif sur les études féministes, les *Gender Studies* et les *Queer Studies*, quant à l'analyse du caractère performatif des catégories de genre. Dans cette perspective, le féminin et le masculin ne sont pas des caractéristiques essentielles d'une personne ou des ensembles d'attributs. Le genre est plutôt un « faire », une performation, qui tire sa force performative de la réitération continuelle du discours et des pratiques liés à ces catégories : « Gender is itself a kind of becoming or activity [...]. Gender ought not to be conceived as a noun or a substantial thing or a static cultural marker, but rather as an incessant and repeated action of some sort » (Butler, 1990, p. 112; Cité par Barad, 2003, p. 208). Les catégories de genre n'ont donc pas de qualités a priori et essentielles, mais sont plutôt performées par leur réitération continuelle. Cette perspective a fait l'objet de nombreuses critiques, dont la faible prise en compte de la matérialité (comme le sexe biologique) et, à l'inverse, la trop grande force attribuée au discours et au langage dans la performativité (Callon, 2007; Mol, 2002; Denis, 2006).

Selon Denis, on peut également attribuer à cette « famille de pensée » différentes recherches qui se sont attardées à la notion de performativité pour examiner les organisations (Taylor et Van Every, 2000; Gramaccia, 2001). Dans cette perspective, Denis écrit : « les organisations ne sont jamais des entités « données », identifiées une bonne fois pour toutes : elles sont « performées » par d'innombrables échanges et transactions quotidiennes » (Denis, 2006).

Finalement, une dernière série de travaux concerne l'analyse des performativités scientifiques et juridiques, et s'inscrit également dans la mouvance de la théorie de l'acteur-réseau. Denis (2006) écrit d'ailleurs qu'un véritable « programme performatif » s'est constitué à cet effet et trouve sa source dans l'ouvrage de Callon sur la performativité de l'économie (Callon, 2006a; Callon, 2007). Cet ouvrage montre la manière dont les travaux de sciences économiques ne se limitent pas à décrire le fonctionnement des marchés, mais participent aussi à la fabrication d'outils (indices boursiers, etc.) qui performent ces mêmes marchés. Ces travaux sur la performativité seront abordés plus loin dans le chapitre.

Bien plus qu'un concept, la performativité est aujourd'hui appréhendée dans bon nombre d'études comme un véritable programme intellectuel visant à contrer les postures

représentationalistes et essentialistes. Comme Denis l'explique, le programme performatif est éminemment politique :

> La critique que rend possible le nouveau programme performatif est profondément pluraliste. Elle consiste à remettre en question la nature des éléments qui ont été assemblés (et souvent l'assemblage lui-même), afin de montrer que d'autres mises en forme sont possibles. Mais ces nouvelles mises en forme ne pourront pas sortir du néant, ni être imposées d'elles-mêmes. Elles devront à leur tour être le fruit d'un long travail de performation : être discutées, expérimentées, autrement dit subir de nombreuses épreuves (Denis, 2006).

2.1.5 En arrière-plan : la philosophie de la technique de Simondon

Un dernier axe théorique, qui pourrait être qualifié d'« arrière-plan » de notre réflexion, concerne la pensée de Simondon, et en particulier son ouvrage le plus connu, *Du mode d'existence des objets techniques* (Simondon, 1958). Le projet (politique) de Simondon, explicité dans les premières pages de son ouvrage, consiste à susciter une prise de conscience du sens des objets techniques qui permettrait de les réintégrer dans la culture. Dans cet essai écrit en 1958, dans un contexte marqué par le déterminisme technique et une technophobie ambiante, Simondon note le malaise de la civilisation occidentale par rapport aux techniques, malaise qu'il attribue à l'opposition dressée entre culture et technique et au postulat selon lequel la technologie ne contient pas de réalité humaine. Les objets techniques, contrairement à d'autres objets culturels comme les œuvres d'arts, sont appréhendés dans notre monde comme s'ils étaient dénudés de sens. La culture reconnaît ainsi certains objets comme porteurs de sens mais refoule d'autres objets, notamment les objets techniques, dans le monde de ce qui ne possède pas de significations mais seulement un usage, une fonction utile.

Plutôt que de poser le regard sur leurs seules dimensions utilitaires ou instrumentales, Simondon propose d'analyser très finement les relations que les objets techniques entretiennent entre eux et avec leur environnement, en portant le regard sur leur évolution. Le terme de « concrétisation », central dans sa pensée, désigne l'ensemble des transformations de l'environnement et de l'objet technique qui mènent à son aboutissement. Par le processus de concrétisation, le dispositif technique établit une continuité entre l'humain et la nature : l'objet technique, ou la machine, constitue donc une médiation entre l'humain et la nature et entre les humains entre eux. En axant le regard sur ce processus de concrétisation, Simondon cherche à porter le regard à « l'intérieur » de l'objet technique, sur son contenu, qu'il définit comme un

mélange stable d'humain et de naturel. Plus que par des caractéristiques fonctionnelles comme son coût ou son fonctionnement, Simondon cherche donc à appréhender l'objet technique pour lui-même, en tant qu'il intègre à la fois un contenu naturel et un contenu humain.

La philosophie de Simondon ne se limite pas à ces analyses de la technique, mais pourrait être décrite, de façon plus fondamentale, comme une philosophie de l'ontogenèse, c'est-à-dire une pensée de la genèse de l'être comme individu. Simondon considère que l'individuation ne produit pas seulement et pas complètement l'individu, mais plutôt un individu relatif, associé à son milieu. Il s'agit donc de saisir l'individu à travers l'individuation plutôt que l'individuation à travers l'individu :

> L'individu serait alors saisi comme une réalité relative, une certaine phase de l'être qui suppose comme elle une réalité préindividuelle, et qui, même après l'individuation, n'existe pas toute seule, car l'individuation n'épuise pas d'un seul coup les potentiels de la réalité préindividuelle, et d'autre part, ce que l'individuation fait apparaître n'est pas seulement l'individu, mais le couple individu-milieu (Simondon, 2007, p. 12).

Par le concept d'individuation, Simondon pose plutôt le regard sur l'opératoire, sur les opérations par lesquelles l'individu, qu'il soit technique ou autre, se concrétise. La pensée de Simondon ne se réduit pas à expliquer la réalisation des individus; il s'agit plutôt d'une pensée préindividuelle, une pensée sur la façon dont la genèse de l'individu constitue en elle-même une relation avec le monde. Il n'y a donc pas, chez Simondon, de réalité individuelle, mais il existe plutôt des singularités qui sont distinctes, mais toujours relatives à autre chose et qui sont comme une rupture dans un équilibre entre des systèmes surtendus, chargés de potentialités (Debaise, 2005). L'individuation est donc toujours partielle et entrelace de façon changeante des aspects préindividuels et des aspects effectivement singuliers.

Simondon consacre également de très belles pages aux rapports entre l'esthétique et la technique qui nous paraissent pertinentes pour comprendre certains des enjeux soulevés par nos analyses subséquentes. Pour Simondon, l'esthétique n'est pas qu'une question de culture et ne peut donc pas être appréhendée uniquement par la seule perception. L'esthétique est plutôt le produit d'une tendance à la perfection dans la médiation entre l'humain et le monde. L'impression esthétique apparaît dans le rattachement au monde, après son détachement. Elle est le lieu de la médiation entre l'humain et le monde, elle devient elle-même un monde, la

structure du monde : « Le caractère esthétique d'un acte ou d'une chose est sa fonction de totalité, son existence, à la fois objective et subjective, comme point remarquable. Tout acte, toute chose, tout moment ont en eux une capacité de devenir des points remarquables d'une nouvelle réticulation de l'univers » (Simondon, 2001, p. 181). Pour Simondon, les objets techniques ne sont pas directement beaux en eux-mêmes, car le caractère de l'objet technique est d'être détaché du monde, au contraire de l'objet esthétique, dont la caractéristique est d'être attachée au monde. Cependant, l'objet technique actualise son potentiel esthétique dans la mesure où il s'insère dans et prolonge le monde. Sa beauté apparaît comme une « insertion des schèmes techniques dans un univers, aux points-clefs de cet univers » (Simondon, 2001, p. 186).

Comme nous le verrons à la fin de ce chapitre, la pensée de Simondon trouve un certain écho parmi certains des auteurs contemporains que nous abordons dans ce chapitre. Pour l'instant, retenons toutefois ici cette insistance de Simondon pour le sens des objets techniques, pour ce qui en fait non seulement des instruments, mais également des médiateurs entre l'humain et le naturel, porteurs de sens et valeurs.

2.2 Concepts clés

Dans cette deuxième partie, nous faisons référence aux travaux mentionnés précédemment pour approfondir certains termes et concepts qui guident notre analyse. Mentionnons à nouveau qu'il ne s'agit pas de développer un modèle théorique explicatif. La démarche que nous proposons ici s'apparente davantage à la théorie de l'acteur-réseau : elle vise en quelque sorte à proposer un vocabulaire qui permet de définir l'orientation de notre regard.

2.2.1 La performativité des artefacts

L'une des questions qui traversent autant les études de la communication que les études STS concerne ce que les artefacts « font » ou « font faire ». Dans les études de communication, ce questionnement sur le rôle des artefacts a été longtemps marqué par une approche fortement déterministe où il existe une relation causale entre l'émergence d'une technologie et certains changements sociaux (par exemple, l'imprimerie qui aurait été la cause de la révolution industrielle). Plus récemment, un déterminisme qui pourrait être qualifié de plus « doux » s'est développé dans le monde anglo-saxon à la suite de l'essai « Les artefacts font-ils de la

politique? » de Langdon Winner (1980; 2002) où l'auteur affirme que les artefacts sont politiques dans ce sens qu'ils peuvent incorporer différentes formes de pouvoir et d'autorité. C'est dans le même sens que Lawrence Lessig (2006), que nous avons déjà cité au chapitre précédent, cherche à appréhender la manière dont le code (informatique) agit à la manière d'une loi.

Les études STS, et en particulier la théorie de l'acteur-réseau, cherchent également depuis longtemps à appréhender cette capacité d'action (*agency*) des artefacts. Le principe de la symétrie généralisée entre humains et non-humains permet en effet de reconnaître la capacité d'agir des non-humains, tels que les objets ou les faits scientifiques. Plus spécifique et d'inspiration sémiotique, la notion d'inscription est utilisée dans la théorie de l'acteur-réseau pour exprimer comment l'objet technique définit entre les acteurs et l'environnement un cadre qui délimite l'action de chacun : « Il s'agit plutôt de faire se spécifier conjointement et de manière indissociable le dispositif technique et son environnement » (Akrich, 1993, p. 92), l'environnement étant défini ici comme les formes de l'organisation sociale, l'environnement naturel et le jeu des acteurs. Certains éléments des dispositifs techniques peuvent n'avoir d'autres fonctions que de signifier à l'usager son niveau d'engagement ou de le contraindre à certaines actions. C'est le cas des demandes de mots de passe, des boutons « j'accepte » ou « valider » que l'on voit aujourd'hui sur de nombreux sites web, qui forcent l'usager à accepter les conséquences de son choix. Ces particularités du design technique impliquent que l'usager soit situé dans un cadre sociotechnique plus large, qui inclut notamment un ensemble de savoir-faire et de normes culturelles et sociales (ce qu'est un contrat, les conséquences légales d'un refus, les connaissances linguistiques). Akrich parle dans ce cas de l'inscription de l'environnement dans le dispositif technique qui contient ainsi un « programme d'action » et contraint en quelque sorte l'activité de l'usager (Akrich, 1993).

La notion de performativité est de plus en plus utilisée pour appréhender cette capacité d'action des artefacts. Dans un court article en art médiatique, Inke Arns propose d'utiliser la notion de performativité pour saisir les conséquences politiques, esthétiques et sociales du code informatique : « This notion – borrowed from speech act theory – not only involves the ability to generate in a technical context, but also encompasses the implications and repercussions of code in terms of aesthetics, politics and society » (Arns, 2005, p. 1). Sur le plan politique, l'auteure réfère à Lessig en écrivant que le code informatique devient la loi, en

ce sens que sa « performativité codée » (« *coded performativity* ») a la capacité de mobiliser ou d'immobiliser les utilisateurs. La spécificité du code informatique, par rapport aux autres actes de langage décrits par Austin, est que les mots correspondent directement au faire : « It directly affects, and literally sets in motion, or even "kills", a process » (Arns, 2005, p. 7). Citant Butler (1997) pour qui un énoncé n'est performatif que s'il produit un effet, Arns soutient que, sur le plan pragmatique, le code informatique n'est performatif que s'il produit effectivement un effet et s'il est exécutable. L'auteure nuance toutefois ce propos en indiquant que, dans le contexte de l'art logiciel, le code non-exécutable a aussi sa raison d'être et pourrait, en lui-même, avoir une certaine force performative.

Adrian Mackenzie (2005) que nous avons déjà cité en problématique, a également consacré un article à la question de la performativité du code informatique. Citant Latour, mais se situant davantage dans une approche *Cultural Studies*, Mackenzie fait surtout référence à Butler (1997) pour expliciter son concept de performativité. Pour Butler, si un énoncé performatif réussit, c'est parce qu'il « fait écho à des actions antérieures et *accumule la force de l'autorité à travers la répétition ou la citation d'un ensemble de pratiques antérieures qui font autorité*[43] » (Butler, 2004, p. 80). S'appuyant sur cette approche, Mackenzie propose de considérer la programmation informatique comme une pratique continuelle de citation du code. Le code informatique tirerait sa force performative de la répétition de ce qu'il nomme l'*authorizing context* (que nous traduisons ici par « contexte d'autorisation ») qui inclurait un ensemble de pratiques, d'imaginaires et d'objets techniques davantage stabilisés. Mackenzie s'est attardé à l'analyse du cas de Linux, un système d'exploitation souvent qualifié de « clone » de Unix, un système d'exploitation plus ancien, développé dans les années 1970. Il remarque que le « Unix » dont Linux serait le clone n'est pas tant un logiciel précis aux frontières bien délimitées mais plutôt un ensemble de pratiques d'administration informatique, de programmation et de design logiciel quelques fois désignées par le terme de « philosophie Unix » (Mackenzie, 2005, p. 84). Le nom LinuX, se terminant par la lettre X, est d'ailleurs une référence directe à cette philosophie UniX. L'auteur propose donc de considérer la « philosophie Unix » comme une partie du « contexte d'autorisation » qui sous-tend le développement de Linux. Mackenzie rappelle que les pratiques de l'informatique qui

43 Les italiques sont dans le texte.

forment ce « contexte d'autorisation » sont également articulées à des pratiques d'exclusion et de hiérarchisation, en particulier en ce qui concerne les rapports de sexe et de genre. Ainsi, dans son courriel qui annonçait la création de Linux, le créateur de ce logiciel faisait référence au « temps mémorable » où « les hommes étaient de vrais hommes qui programmaient leur propre gestionnaire de périphériques » (Mackenzie, 2005, p. 87). La performativité du code est donc également une performativité du genre.

Comme indiqué plus tôt dans ce chapitre, c'est cependant davantage vers le développement récent de la performativité dans les études STS et en anthropologie de l'écriture que nous souhaitons nous tourner. La référence à la pragmatique du langage et à la notion de performativité est implicite depuis déjà longtemps dans les études STS (Licoppe, 2010). Sans le citer, Bruno Latour fait par exemple une allusion directe à l'ouvrage de Austin, *Quand dire c'est faire* (1991), lorsqu'il décrit la capacité d'agir de certains artefacts comme les programmes ou les puces informatiques :

> Les programmes s'écrivent, les puces se gravent comme des eaux-fortes, ou se photographient comme des plans. Et pourtant **ils font ce qu'ils disent** ? Mais oui, car tous, textes et choses, ils agissent. Ce sont des programmes d'action dont le scripteur délègue la réalisation tantôt à des électrons, tantôt à des signes, tantôt à des habitudes, tantôt à des neurones (Latour, 1992a, p. 182)[44]

Parmi les travaux ayant abordé la performativité des artefacts, ceux réalisés dans la perspective d'une anthropologie de l'écriture méritent une attention particulière (Fraenkel, 2007; Fraenkel, 2006; Fraenkel, 2008; Denis et Pontille, 2010a; Denis et Pontille, 2010b; Denis, 2006). D'une part, parce que ces travaux nous ont particulièrement éclairés dans notre compréhension de la performativité. D'autre part, car l'objet de ces travaux, l'écrit, résonne avec notre propre objet d'étude, le code source, en tant qu'il constitue un artefact écrit.

Denis et Pontille, dans leur étude sur la signalétique du métro de Paris (Denis et Pontille, 2010b), soutiennent ainsi que la performativité des écrits, les panneaux du métro dans leur cas, repose sur un travail de maintenance qui assure à ces écrits une permanence et une stabilité dont ils ne sont pas pourvus naturellement. Pour qu'un panneau de signalisation soit performatif, c'est-à-dire pour qu'il ait l'effet désiré d'orienter les voyageurs dans le métro, celui-ci doit notamment être conforme à certaines normes graphiques et être situé à des

44 Nous soulignons.

endroits significatifs. Or, ces différentes propriétés ne sont pas intrinsèques à la matérialité des panneaux, mais reposent au contraire sur un incessant travail invisible qui vise à concevoir et à maintenir l'exposition de ces écrits. L'apparente immuabilité et la stabilité de ces objets sont donc basées en bonne partie sur un travail en coulisse. La performativité de l'écrit, de la signalétique dans le cas qui les concerne, repose donc sur « la mise en invisibilité des agencements fragiles qui la rendent possible et du travail incessant qu'elle nécessite » (Denis et Pontille, 2010a, p. 126).

Précurseure dans cette réflexion sur l'écrit, Béatrice Fraenkel (2006; 2007; 2008) dresse pour sa part les contours d'une anthropologie pragmatique de l'écriture, qui s'intéresse à des actes courants d'écriture, comme la signature ou l'étiquetage, qui participent à la performativité d'un acte écrit. Dans un article où elle critique la place de l'écrit dans la théorie de la performativité d'Austin, cette auteure analyse le testament, un cas important de la théorie de la performativité d'Austin. Contrairement à ce que soutient Austin, pour qui l'effectuation de la performativité se situe à la lecture du testament, ou à son exécution, Fraenkel soutient que la performativité réside également dans l'agencement de l'acte écrit dans une chaîne d'écriture plus large. Citant Latour (2004), l'auteure note par ailleurs que les travaux empiriques sur la fabrique du droit montrent « qu'il existe une relation directe entre la force exemplaire de l'acte juridique et le réglage précis de ses formes textuelles, graphiques, matérielles » (Fraenkel, 2006). La performativité d'un testament repose donc sur le travail du ou de la notaire (et de ses assistant-es) qui doit donner une cohérence juridique à ces dernières paroles, en mettant en forme l'acte, et en apposant les sceaux et les signatures appropriés. Le testament, en tant qu'acte écrit, doit, pour être performatif, « être inséré dans un système de chaînes d'écriture, de personnes habilitées et de signes de validation, l'ensemble de ces éléments forment l'authenticité nécessaire à la performativité » (Fraenkel, 2006) C'est en bonne partie cette mise en forme, cette codification de l'acte juridique, qui donne une force performative au testament, et non pas seulement sa lecture ou son exécution. Fraenkel pose ainsi l'hypothèse que « tous les phénomènes de mise en forme jusqu'aux choix typographiques sont susceptibles de porter des significations et de participer à l'effectuation d'un énoncé performatif » (Fraenkel, 2006). Ce sont des actes d'écriture, telles l'apposition de signatures ou de tampons, qui confèrent à l'écrit une certaine matérialité nécessaire à sa performativité. Fraenkel en appelle en outre à analyser des actes d'écriture, « comme ceux

désignés par les verbes : copier, enregistrer, signer, étiqueter, afficher, etc. » (Fraenkel, 2006), en analysant leur force propre.

Ces différentes réflexions permettent d'orienter notre regard sur le code source. En particulier, l'analyse que fait Fraenkel de la performativité des actes juridiques résonne avec la métaphore législative mobilisée par Lessig (2006) (présentée en problématique) pour décrire la force régulatrice du code informatique. Fraenkel (2006) note également que l'observation des activités d'écriture permet de pénétrer au cœur d'une institution ou, pourrions-nous ajouter dans le cas de notre étude, des infrastructures technologiques basées sur le code informatique. Suivant cette analyse, il s'agit donc de s'intéresser à des actes concrets d'écriture, tels la signature ou l'étiquetage, qui confèrent au code source la stabilité nécessaire à sa force régulatrice.

2.2.2 L'artefact comme assemblage

La notion d'artefact est amplement mobilisée par les auteurs que nous avons cités plus tôt. Dans un article sur la règle et son rappel, Denis (2007) utilise la notion d'artefacts *prescriptifs* pour référer à ces artefacts qui inscrivent directement dans l'environnement de travail des dimensions de la prescription. Ailleurs, Denis et Pontille (2010a; 2010b) font référence à des artefacts *écrits* ou encore à des artefacts *graphiques* pour désigner les panneaux de signalisation. Pour ces auteurs, ces artefacts constituent en fait des artefacts *hybrides* dont le caractère graphique et langagier compte autant que leur matérialité et leur position dans l'environnement. Laztko-Toth développe pour sa part le concept d'artefact *numérique* pour appréhender un artefact de nature logicielle (ou numérique), qui est « essentiellement constitué » (Latzko-Toth, 2010, p. 47) de code informatique. Pour Laztko-Toth, l'artefact numérique est également un artefact de type hybride, en ce sens qu'« il relève à la fois des registres matériel et symbolique » (Latzko-Toth, 2010, p. 47)[45].

Suchman (2007) utilise pour sa part la notion d'artefact *computationnel* (*computational artefact*), sans que celle-ci ne soit véritablement définie. Cette notion pourrait être traduite par celle d'artefact numérique, ou d'artefact informatique, mobilisée par Latzko-Toth (2010).

---

45 Une autre catégorie bien connue est celle de l'artefact cognitif, catégorie mise de l'avant par Don Norman (1993) pour décrire certains types d'artefacts ayant une dimension cognitive.

Toutefois, la notion d'artefact computationnel mobilisée par Suchman semble renvoyer aux travaux en intelligence artificielle et caractériser un certain caractère « animé » de l'artefact. Il nous semble donc plus pertinent ici d'adopter la traduction plus directe et littérale d'artefact computationnel pour retenir la référence au concept de computationalité qui renvoie à l'idée d'un artefact « animé » par la computation (c'est-à-dire un programme informatique en exécution), au contraire d'un document ou d'une base de données qui, quoique basés sur le code informatique, ne sont cependant pas eux-mêmes animés.

Sans refaire le travail de définition réalisé par Latzko-Toth dans sa thèse (Latzko-Toth, 2010, p. 43-49), nous souhaitons ici établir quelques précisions conceptuelles sur la notion d'artefact[46]. Selon *Le Petit Robert*, la notion d'artefact renvoie à la fois à un « phénomène d'origine humaine, artificielle », mais également à une « altération produite artificiellement lors d'un examen de laboratoire » (« Artefact », *Le Petit Robert 1*). Une troisième définition, dont ne fait pas mention *Le Petit Robert*, appréhende l'artefact comme un produit dérivé d'une recherche scientifique ou d'un développement technique.

À la suite de ce tour d'horizon des différents usages de la notion d'artefact, il semble important de nous questionner sur le type d'artefact que constitue le code source informatique. En particulier, le code source constitue-t-il un artefact « computationnel » ou un artefact « numérique » ? Selon la définition que donne Latzko-Toth (2010) des artefacts numériques, ceux-ci auraient la particularité d'être constitués de code informatique. Mais le code informatique constitue-t-il lui-même un artefact numérique ? Et qu'en est-il du code source ? Suivant Suchman (2007), nous pourrions assumer qu'une des propriétés de l'artefact computationnel est justement sa computationalité, le fait d'être « animé » informatiquement. Or, de façon assez paradoxale, le statut du code source, comme artefact, serait sans doute plus près de celui d'un document de traitement de texte que d'un logiciel, puisqu'il est en soi « inerte ».

Il n'est cependant pas question ici de tenter une purification du concept d'artefact, ou de l'un de ses substantifs (écrit, cognitif, informatique), mais plutôt d'interroger le statut pragmatique

46 Nous ne ferons pas ici une revue extensive des différentes définitions de ce concept, ce qui a par ailleurs déjà été fait par Latzko-Toth (2010) dans sa thèse.

du code source en regard d'autres types d'artefacts, notamment des artefacts numériques, qui exploitent les technologies du traitement de l'information.

Latzko-Toth (2010) note que la notion d'artefact se retrouve en abondance dans les études sur les sciences et les technologies, sans que cette notion ne soit jamais très bien définie au niveau conceptuel. Il remarque par exemple qu'elle est absente du lexique de la théorie de l'acteur-réseau proposée par Akrich et Latour (1992). D'ailleurs, Suchman, bien qu'elle mobilise également abondamment la notion d'artefact sans la définir formellement, cite toutefois l'anthropologue Alfred Gell (1998) qui définit les artefacts comme étant « those objects taken to be instruments or outcomes of social agency » (Suchman, 2007, p. 244). Même si les études STS et la théorie de l'acteur-réseau en particulier sont réticentes à préciser la notion d'artefact, il nous semble que leur usage de cette notion correspond surtout au sens général d'une *chose fabriquée.* Retenons donc tout d'abord cette première définition pour notre thèse. Il s'agit donc de nous intéresser à la *fabrication* de l'artefact.

Cette manière de définir la notion d'artefact reste cependant problématique. Pour Latzko-Toth (2010), l'absence de conceptualisation explicite concernant la notion d'artefact s'expliquerait par le souci de ne pas briser le principe de symétrie généralisée en insistant trop sur le caractère artificiel de l'artefact :

> Cette absence peut s'expliquer du fait que cette approche théorique tend au contraire à minimiser la distance entre le naturel et l'artificiel, notamment en désignant les entités artificielles par les termes d'actant et de « non-humain ». Accorder le statut de concept à la notion d'artefact reviendrait donc à inscrire dans le langage descriptif de cette sociologie, une asymétrie fondamentale entre les actants qu'elle s'ingénie à abolir (Latzko-Toth, 2010, p. 44).

Citant les travaux de Callon (2006b), Latzko-Toth (2010, p. 47) remarque que l'artefact est appréhendé dans la théorie de l'acteur-réseau au même titre que le fait scientifique. Alors qu'ils sont perçus par les acteurs comme des « choses » aux frontières bien définies, les artefacts, comme les faits scientifiques, seraient plutôt des assemblages hétérogènes d'actants humains et non-humains (Callon, 2006b, p. 272; Latzko-Toth, 2010, p. 47). Inspirée de la théorie de l'acteur-réseau, on retrouve une approche similaire chez Denis qui écrit qu'« au même titre que d'autres entités, les artefacts écrits prennent non seulement leur signification à partir de la relation qu'ils entretiennent avec les autres, mais le caractère achevé de leur

forme est lui-même une conséquence directe de cette relation » (Denis et Pontille, 2010a, p. 110).

La notion d'*assemblage* est elle aussi centrale dans les travaux en STS. Comme nous l'avons mentionné, dans la théorie de l'acteur-réseau, l'artefact est appréhendé comme un assemblage hétérogène d'actants humains et non-humains. Cette notion est notamment mobilisée par Suchman (2007) pour faire contrepoids à celle du Cyborg, mise de l'avant vers la fin des années 1990 dans le très populaire « Manifeste Cyborg » de Donna Haraway (1985). Pour Suchman, si la figure du Cyborg de Haraway permet de réfléchir à des formes inédites de socialité pouvant émerger de la rencontre entre humains et machines, cette figure tend à occulter les difficultés à assembler l'humain et le machinique. Suchman donne les exemples du travail nécessaire à la réalisation et à la mise en place d'une prothèse cardiaque, et des douleurs de dos inhérentes à la pratique de l'informatique (Suchman, 2007, p. 275). Elle considère également que la figure du Cyborg hérite d'un désavantage propre à toute stratégie centrée sur une figure héroïque, soit la négligence des sites plus banaux de reconfiguration des relations entre humains et machines. Suchman propose ainsi de situer la figure de l'assemblage en continuité de la figure hybride du Cyborg. Au contraire de la figure du Cyborg, qui laisse supposer une identité stable et sans couture, l'assemblage met de l'avant l'idée d'une certaine complexité et d'une difficulté à associer les composantes humaines et machiniques (ou non-humaines, de façon plus générale) : « Expanded out from the singular figure of the human-machine hybrid, the cyborg metaphor dissolves into a field of complex sociomaterial assemblages currently under study within the social and computing sciences » (Suchman, 2007, p. 283). Au contraire de la figure du Cyborg, celle de l'assemblage nous amène à insister davantage sur le travail, souvent invisible, qui participe à la solidification des assemblages et lui donne son apparence de tout cohérent. De la même manière, et comme nous le verrons plus loin, la figure de l'assemblage nous amène également à interroger les multiples configurations, découpages et articulations *à l'intérieur* de ces assemblages.

La figure de l'assemblage est également présente, de manière implicite ou explicite, dans les travaux récents sur la performativité au point où Denis considère que, comme cité plus tôt, la critique permise par le programme performatif consiste « à remettre en question la nature des éléments qui ont été assemblés (et souvent l'assemblage lui-même) » (Denis, 2006). Ces travaux ne mettent effectivement pas seulement l'accent sur l'effet des artefacts, mais plutôt

sur les conditions de leur existence, sur ce qui consolide l'assemblage des entités. Cet aspect est particulièrement développé dans l'article de Bruno Latour publié dans le numéro de la revue *Réseaux* sur la performativité des artefacts, où l'auteur considère l'objet technique comme un assemblage momentané, qui doit être constamment réactualisé, raccordé :

> « On ne trouvera donc jamais le mode d'existence technique dans l'objet lui-même puisqu'il laisse partout des hiatus : d'abord, entre lui-même et le mystérieux mouvement dont il n'est que le sillage ; ensuite, à l'intérieur de lui-même entre chacun des ingrédients dont il n'est que l'assemblage momentané. Il n'y a jamais en technique de solution de continuité ; ça ne "fait jamais raccord" » (Latour, 2010, p. 25).

Latour note en effet que, de la même manière dont on oublie d'ajouter à la connaissance objective ses chemins de référence, on oublie souvent également de prendre en compte ce qui instaure et assure la vitalité des objets techniques : « On omet toujours d'ajouter aux objets techniques ce qui les instaure sous prétexte, ce qui est vrai aussi, qu'ils se *tiennent tout seuls* une fois lancés, sauf qu'ils ne peuvent jamais demeurer seuls et sans soin – ce qui est vrai aussi » (Latour, 2010, p. 26). Pour Latour, « l'objet technique a ceci d'opaque et, pour tout dire, d'incompréhensible, qu'on ne peut le comprendre qu'à la condition de lui ajouter les *invisibles* qui le font exister d'abord, puis qui l'entretiennent, le soutiennent et parfois l'ignorent et l'abandonnent » (Latour, 2010, p. 26). En d'autres termes, pour que l'artefact puisse exister et « performer », il faut que son assemblage soit également continuellement performé.

2.2.3 Reconfigurations

Le concept de reconfiguration est au centre de l'approche développée par Suchman dans son ouvrage *Human-Machine Reconfigurations* (Suchman, 2007). Celui de configuration est également bien présent dans d'autres travaux. Sans en faire un concept, Serge Proulx fait par exemple usage de la notion de configuration lorsqu'il écrit que la configuration technique et linguistique du dispositif constitue une « programmation » des possibilités d'usage (Proulx,

2002)[47]. Ailleurs, Proulx mobilise ce concept de façon explicite lors qu'il identifie comme l'un des niveaux de sa théorie des usages, « L'inscription de dimensions politiques et morales dans le design de l'objet technique et dans la configuration de l'usager » (Proulx, 2005b, p. 10). Citant notamment le texte de Woolgar (1991), sur lequel nous reviendrons plus loin, Proulx soutient que certaines valeurs, notamment celles liées à la rationalité technique, sont inscrites dans les dispositifs de communication et que, de la même manière, des rapports sociaux sont contenus dans le design même de l'objet technique.

Comme mentionné plus tôt, la notion de configuration est également mobilisée dans le texte de Hess (2001) lorsqu'il soutient que les études STS de deuxième génération tendent à privilégier certaines *configurations* plutôt que d'autres (« For example, what alternatives are there to the current configuration of the production [...] ») (Hess, 2001, p. 10). Finalement, Suchman mobilise également la notion de configurations pour définir son concept plus général de reconfigurations, dans un sens très près de celui décrit précédemment par Hess (2001) :

> One form of intervention into current practices of technology development, then, is through a critical consideration of how human and machines are currently figured in those practices and how they might be figured – and configured – differently (Suchman, 2007, p. 227).

Dans le dernier chapitre de son ouvrage, Suchman affirme adopter une perspective développée dans les travaux récents des études féministes et culturelles de la science, à savoir, une attention aux questions de *figurations*. La question de la figuration apparaît dans les travaux féministes, et en particulier dans ceux d'Haraway (1997, p. 11), pour proposer que tous les langages soient figuratifs, y compris ceux plus techniques et mathématiques, dans ce sens que leurs éléments invoquent des associations entre divers niveaux de sens et de pratiques. Les technologies pourraient ainsi être considérées comme des *figurations matérialisées* (« materialized figuration ») (Suchman, 2007, p. 227), car elles constituent des

---

47 Dans cet article, dont l'objet concerne principalement l'identité québécoise, Proulx (2002) considère que les technologies de l'information amènent un nouveau rapport au monde qui peut ébranler les assisses identitaires du sujet québécois mais qui, en même temps, ouvre sur une fourchette de possibilités cognitives et pratiques. Dans le contexte où se trouve affaibli le poids de la langue française comme référent identitaire principal des Québécois et Québécoises, Proulx se demande également s'il ne serait pas important de repenser sur d'autres bases les traits qui constituent l'identité québécoise.

assemblages plus ou moins stables de choses et de sens, d'humain et de matériel. Pour Suchman, les effets de la figuration sont politiques car les discours, images et normativités spécifiques qui informent les pratiques de figurations peuvent soit réinscrire des ordonnancements sociaux existants, soit les remettre en question.

Comme décrit dans le passage cité plus haut, selon Suchman, une intervention critique dans le développement technologique consisterait à comprendre la façon dont les humains et les machines sont figurés dans ces pratiques et la façon dont des figures alternatives peuvent potentiellement remettre en question les régimes actuels de production scientifique et technologique (Suchman, 2007, p. 228). Pour Suchman, cet effort participe également d'un objectif plus grand – qui nous semble d'ordre épistémologique – consistant à appréhender la science comme une culture, c'est-à-dire à penser la science non pas comme une démarche de découverte de lois universelles, mais plutôt comme : « l'élaboration continuelle et la transformation potentielle de pratiques historiquement et culturellement spécifiques, dans lesquelles nous sommes tous et toutes impliqués plutôt que seulement témoins modestes »[48] (Suchman, 2007, p. 227). Concernant ce dernier concept de « témoin modeste », mentionnons que Suchman fait ici référence aux travaux de Haraway (1997), Latour (1993) et Shapin et Schaffer (1985), ce qui montre bien l'ancrage de cette perspective dans les études STS (Suchman, 2007, p. 227).

Soulignons ici que l'intérêt empirique de Suchman dans cet ouvrage concerne plus spécifiquement les figures conjointes de l'humain – pris au sens d'humanité – et de la machine. S'appuyant notamment sur les études qu'elle a réalisées concernant la recherche en intelligence artificielle, elle note que la figure prévalente de l'humain dans l'imaginaire euro-américain est celle d'un agent rationnel et autonome, figure que les recherches en intelligence artificielle perpétuent :

> At stake, then, is the question of what other possible conceptions of humanness there might be, and how those might challenge current regimes of research and development in the sciences of the artificial, in which specifically located individuals conceive technologies made in their own image, while figuring the latter as universal (Suchman, 2007, p. 228).

48 « [...] the ongoing elaboration and potential transformation of culturally and historically specific practices, in which we are all implicated rather dans modest witness » (traduction libre).

L'objet d'analyse de Suchman – les figures conjointes de l'humanité (*humanness*) et de la machine – n'est pas le nôtre. Notre intérêt se restreint plutôt à la manière dont certains dispositifs particuliers, de même que leurs usagers, sont figurés et reconfigurés, en prenant notamment en compte l'articulation de certaines valeurs dans ces reconfigurations. Au niveau de l'objet d'analyse, le texte de Woolgar (1991) est sans doute plus près du nôtre. Woolgar propose de considérer dans cet article la manière dont la conception technologique participe d'une *configuration de l'usager*. S'appuyant sur une ethnographie qu'il a réalisée dans une firme de production de micro-ordinateurs (durant les années 1980), il note que la définition de ce qui constitue la machine est intimement liée à la définition du contexte de la machine, et en particulier, de cet aspect du contexte nommé « l'usager » :

> In other words, representations (descriptions, determinations of many kinds) of « what the machine is » take their sense from descriptions of « the machine's context » ; at the same time, an understanding of « the context » derives from a sense of the machine in its context. The sense of context and machine mutually elaborate each other (Woolgar, 1991, p. 66).

Pour Woolgar (1991), la configuration de l'usager implique donc un certain travail de démarcation des frontières (p. 89). Le profil et la capacité d'action du futur utilisateur sont structurés et définis dans sa relation avec la machine. Dans une certaine perspective, la manière dont l'usager est figuré dans la conception de la machine s'inscrit dans son design même qui, à son tour, participe d'une certaine configuration de l'usager.

2.2.4 Frontières, démarcations et interfaces

Un autre aspect conceptuel de notre cadre d'analyse concerne la famille de concepts autour des notions de frontières, de démarcations et d'interfaces. La notion de frontière a été particulièrement mobilisée en STS, notamment par le concept d'objet-frontière (Star et Griesemer, 1989), utilisé pour décrire des « entités qui servent "d'interface" (Fujimura, 1992) dans la coopération entre des acteurs ayant des perspectives différentes » (Latzko-Toth, 2010, p. 42). Ces notions sont également mobilisées pour décrire le travail de démarcation qui délimite par exemple la science et la non-science (Gieryn, 1983).

C'est cependant la conception performative de la frontière et de l'interface, développée par Suchman, que nous souhaitons explorer ici. Suchman s'inspire, dans sa conceptualisation de la frontière, de l'approche du « réalisme agentiel » développée par la philosophe féministe des

sciences Karen Barad, qui implique d'appréhender la réalité comme la « sédimentation du processus consistant en la fabrication d'un monde intelligible, à travers certaines pratiques plutôt que d'autres » (Barad, 2003, p. 105). À travers une lecture de Niels Bohr, Barad (2003) insiste sur l'inséparabilité de « l'objet observé » et des « appareillages d'observation » (*agencies of observation*) de cet objet. S'inscrivant à la suite de Haraway, pour qui ce qui est vu est intimement lié à ce qui voit, Barad soutient que différents « appareillages d'observation » entraînent différents découpages dans la constitution des objets. En d'autres termes, la constitution des objets est toujours le résultat d'un découpage, et ce découpage entraîne certaines conséquences plutôt que d'autres. C'est pour cette raison qu'il est nécessaire de comprendre les effets performatifs d'assemblages particuliers :

> Because the cuts implied in boundary making are always agentially positionned rather than naturally occuring, and because boundaries have real consequences, « accountability is mandatory » (Barad, 2003, p. 187). The accountability involved is not, however, a matter of identifying authorship in any simple sense but rather a problem of understanding the effects of particular assemblages and assessing the distributions, for better and worse, that they perform (Suchman, 2007, p. 285).

Pour Suchman, cette manière de penser les objets implique une conception très différente de l'interface : « [It] reminds us that boundaries between human and machines are not naturally given but constructed in particular historical ways and with particular social and material consequences » (Suchman, 2007, p. 287). L'interface n'est en effet plus une entité établie a priori : elle désigne plutôt certaines formes de découpage qui se produisent dans toutes les pratiques sociomatérielles et qui permettent différentes « intra-actions » sujet-objet. De la même manière, une telle conception de l'interface permet de faire exploser cette catégorie pour saisir la multiplicité des configurations et des rencontres entre humains et machines, *à l'intérieur même* des configurations sociomatérielles.

Suchman cherche dans cette référence à Barad à critiquer la conceptualisation commune et restreinte de l'interface comme une frontière a priori et fixe, par l'intermédiaire de laquelle l'humain et la machine interagiraient. L'objet de sa critique vise principalement les projets de conception d'interfaces « interactives » par le biais desquelles les artefacts computationnels pourraient converser à la manière d'êtres humains (« in a human-like matter »). Pour

Suchman, cette conception de l'interface relève du fantasme d'une infrastructure parfaitement invisible et d'une économie de service sans serviteurs :

> Discourses of agency at the interface at once naturalize the desirability of « service provision » and further obscure the specific sociomaterial infrastructure – including growing numbers of human workers – on which smooth interactions at the interface continue to depend (Suchman, 2007, p. 225).

Citant une recherche de Doyle (1997) sur l'intelligence artificielle, Suchman (2007, p. 215) suggère plutôt de s'intéresser à la manière dont la vitalité et l'apparente autonomie des artefacts computationnels, voire leur « animisme », reposent sur l'assemblage massif de machines, d'utilisateurs et de rhétoriques. Dans cette perspective, plutôt que d'appuyer le projet de construire des interfaces « intelligentes » interagissant de manière « humaine », l'auteure propose une conception étendue de l'interface, vue non pas comme une frontière fixe et a priori entre l'humain et la machine, mais comme un élément dans un « réseau étendu de relations arbitrairement découpées – bien que de manière intentionnée – à travers des actes pratiques, analytiques et/ou politiques de fabrication des frontières » (Suchman, 2007, p. 245).

Ainsi, plutôt que de parler de « conversation à l'interface », Suchman oriente plutôt le regard vers les possibilités offertes par les nouveaux médias pour s'attarder à la manière dont l'interface elle-même est dynamiquement reconstituée dans une relation changeante entre les humains et les machines. Dans une telle conception, la « conversation » entre l'humain et la machine n'est pas tant un lieu d'échange de messages, mais devrait plutôt être appréhendée comme la coproduction continuelle et contingente d'un monde sociomatériel partagé (Suchman, 2007, p. 23). Finalement, plutôt que de recourir à la métaphore de la conversation pour décrire nos relations avec les artefacts computationnels, l'auteure suggère la métaphore de l'écriture et de la lecture pour décrire nos relations avec les artefacts computationnels.

Suivant cette analyse, nous nous intéressons dans le cadre de notre étude à la manière dont le code source constitue également une interface entre l'humain et la machine, et à la manière dont l'écriture et la lecture du code source constituent une forme de conversation, ou d'interaction, entre l'humain et la machine. Nous nous intéressons particulièrement à la manière dont le code source, dans son design même, opère des découpages « sociaux » en privilégiant certaines figures d'acteurs au détriment d'autres.

2.2.5 Contributions et « actes configurants »

La notion de contribution prend une place de plus en plus importante dans le travail et les échanges sur Internet au point où certains auteurs parlent aujourd'hui de la mise en place d'une véritable économie de la contribution qui se distinguerait des formes d'économie basées sur le don ou sur les transactions marchandes (Stiegler, Giffard, et Faure, 2009). Cette « économie de la contribution » serait fondée notamment sur la motivation, l'intérêt du plus grand nombre et une division cognitive du travail (Goldenberg, 2010).

Notre intérêt porte ici cependant plus précisément sur une conceptualisation plus formelle de la contribution en tant qu'acte. Analysant l'usage des messageries instantanées dans des entreprises, Licoppe, Proulx et Cudicio (2010) proposent de considérer le *genre communicationnel*[49] « question rapide » en tant que forme de contribution définie comme « une transaction fondée sur l'échange de messages isolables et discrets (une information, un nom, un numéro de version logicielle), orientée vers le service, l'entraide, la coopération interpersonnelle » (Licoppe, Proulx, et Cudicio, 2010). Analysant la forme contribution, ils notent que celle-ci « serait d'autant plus réussie et significative [...] qu'elle serait façonnée pour demander un effort cognitif minimal au contributeur » (Licoppe, Proulx, et Cudicio, 2010). Pour ces auteurs, la contribution serait une transaction qui ne relèverait ni du don, ni de la transaction marchande. Ils proposent finalement l'hypothèse que la *forme contribution* puisse constituer plus généralement « le fait social total d'un capitalisme cognitif et connecté, fortement irrigué par les médias sociaux » (Licoppe, Proulx, et Cudicio, 2010).

Allant sensiblement dans le même sens, Goldenberg (2010) propose dans sa thèse de doctorat une définition de la notion de contribution qui s'oppose notamment au concept de don développé par le mouvement du Mauss. L'auteure prend notamment ses distances d'une définition du « don » comme souci du lien social, tel que mis de l'avant par Godbout (2000). Pour Goldenberg, ce souci du lien social ne serait pas à la base des échanges « épistémiques » que l'on retrouve dans les communautés de logiciels libres et de wikis (son objet d'étude). Dans ces communautés « épistémiques », le lien social émergerait non pas d'une intentionnalité du contributeur, mais plutôt des relations soutenues entre contributeurs et

---

49 Citant Orlikowski et Yates (1994), les auteurs définissent le *genre communicationnel* comme « l'articulation cohérente entre un type de contenu, une finalité et une forme spécifiques des activités communicationnelles » (Licoppe, Proulx, et Cudicio, 2010, p. 2).

membres du projet, dans leur souci de continuer le travail collectif. Ainsi, citant un article de Richardson (2002), elle note que la contribution dans l'univers du logiciel libre ne serait pas tant liée à un objectif de construction du lien social, mais répondrait plutôt à des besoins et à des intérêts personnels, soit d'adapter le logiciel à ses propres besoins. Dans le cas des wikis, la contribution se distinguerait du don selon plusieurs points : la contribution serait orientée vers la construction et l'usage collectif du bien commun plutôt que vers le simple échange; il n'y a pas de sentiment de dette fort dans la contribution, comme c'est le cas pour le don; la reconnaissance de la contribution serait liée à l'utilité de la prestation. L'auteure pose l'hypothèse que « si le don vise et reconnaît l'autre dans un souci de lien social, la contribution aux wikis publics se présente plutôt comme une participation constitutive à une chose commune » (Goldenberg, 2010, p. 109).

Dans sa thèse de doctorat, Latzko-Toth (2010) cherche pour sa part à esquisser un modèle pour penser les contributions au développement d'un dispositif technique tel que l'IRC (l'objet à l'étude dans sa thèse), en articulant ces contributions à la perspective d'une co-construction du dispositif technique. Dans une perspective proche de celle que nous développons ici, Latzko-Toth soutient que la contribution constitue une redistribution de la capacité d'agir, dans ce sens qu'elle reconfigure la capacité d'action des dispositifs techniques et celle des acteurs humains, en ébranlant notamment la frontière entre concepteur et usager. Cette dernière piste ouvre la voie à une conceptualisation de la contribution propre à l'étude des sciences et des technologies, en ce sens que la contribution ne se restreindrait pas à la production de contenu ou de données, mais consisterait également en une participation à la conception des dispositifs techniques. Cette approche est également très près de celle développée par Suchman (2007), pour qui la *reconfiguration* consiste en une transformation de la manière dont les humains et les machines sont *figurés* dans leurs relations mutuelles et qui se fait à travers des *actes* d'engagement qui sont liés à une certaine responsabilité. Dans le cadre de notre étude, nous voudrions articuler plus explicitement la notion de contribution, telle que mobilisée par les acteurs et dans les écrits, comme un acte de reconfiguration de l'assemblage humain-machine qu'est l'Internet.

2.3 Enjeux moraux et épistémologiques de l'approche choisie

> We are tackling one of the most important, widespread institutions of our time – science and technology. And if this doesn't mean social reform of some sort, what, by our own analysis, does it mean ? [...] Where are our communities and commitments ? What sort of moral order are we building or trying to reform ? (Star, 1995a, p. 25).

Notre réflexion théorique s'est précisée et enrichie durant notre parcours. Au début de notre étude (même avant, si l'on tient compte de la maîtrise), nous avions décidé d'explorer certaines approches qui nous intéressaient, sans toutefois bien comprendre les tenants et aboutissants de ces approches, d'un point de vue épistémologique et moral. Notre démarche pourrait plutôt se décrire comme une prise de conscience progressive des fondements plus philosophiques de ces approches et, par là, de notre propre réflexion théorique. D'ailleurs, cette démarche est probablement en phase avec celle des auteurs cités dans ce chapitre qui, ancrés dans des études d'abord sociologiques et anthropologiques, ont ensuite cherché à élargir leur réflexion vers des perspectives plus générales et philosophiques. Dans cette dernière partie, notre objectif consiste donc à articuler les différents axes conceptuels développés plus tôt dans ce chapitre avec une réflexion plus large d'ordre philosophique. Nous tentons tout d'abord de situer plus explicitement notre démarche, d'abord face à la philosophie de Simondon, et ensuite par rapport au pragmatisme, deux horizons philosophiques auxquels nous pouvons rattacher, avec certaines limites, les travaux présentés dans ce chapitre. Nous approfondissons ensuite deux enjeux qui sont souvent soulevés en critique à ces travaux, soit la question du relativisme, et celle du rôle des non-humains.

2.3.1 Préciser la posture épistémologique

Une première avenue pour situer notre posture consisterait à ancrer notre réflexion au fondement de la pensée de Simondon. La philosophie de Simondon ne se limite en effet pas aux analyses de la technique, mais pourrait être décrite, tel que mentionné précédemment, de façon plus fondamentale, comme une philosophie de l'ontogenèse, soit une pensée de la genèse de l'être comme individu. En introduction de sa thèse, Simondon (1958; 2001) situe l'origine de sa démarche dans son opposition à deux pensées philosophiques classiques, à savoir l'atomisme et l'hylémorphisme. Dans l'atomisme, l'atome est un individu donné, indivisible, à partir duquel émergent des individus composés. Dans l'hylémorphisme, qui est

la doctrine d'Aristote, l'être individuel n'est pas donné, mais se situe plutôt dans la rencontre entre la forme et la matière, dans une « une opération par laquelle une forme préexistante façonne une matière » (Debaise, 2005). Simondon cherche à se détacher autant de l'atomisme, qui postule l'existence d'êtres individuels élémentaires, et de l'hylémorphisme, qui laisse de côté la compréhension précise du rapport s'opérant entre forme et matière. Par le concept d'individuation, Simondon axe le regard sur l'opératoire, sur les opérations par lesquelles l'individu se concrétise. La pensée de Simondon ne se réduit pas à expliquer la réalisation des individus; il s'agit plutôt d'une pensée préindividuelle, une pensée sur la façon dont la genèse de l'individu constitue en elle-même une relation avec le monde.

Plusieurs auteurs abordés dans ce chapitre, en particulier ceux rattachés à la théorie de l'acteur-réseau, font référence à plus d'un endroit à la pensée de Simondon. Ainsi, dans un article qui explore l'intérêt du concept de médiation, Akrich affirme partager le projet de Simondon de redonner leur épaisseur aux dispositifs techniques :

> Il faut redonner aux dispositifs techniques leur épaisseur, ce qui en fait des médiateurs et non de simples instruments ou encore, pour reprendre les termes de Simondon, ce qui en eux-mêmes peut être décrit comme un mixte stable d'humain et de naturel, de social et de matériel; il faut montrer comment se constituent conjointement les techniques et leur environnement social et naturel, ou encore comment, en utilisant à nouveau Simondon, les objets techniques sont à la fois connaissances et sens des valeurs (Akrich, 1993, p. 91).

Bruno Latour (2001) cite également Simondon en définissant la médiation technique comme le moment où « matière et société échangent leurs propriétés » (Latour, 2001, p. 198). Pour Latour, la médiation technique constitue en quelque sorte la zone d'articulation entre l'humain et le non-humain ou encore, selon un style imagé propre à cet auteur, le moment où « Dédale plie, tisse, trame, machine, trouve des solutions là où aucune n'apparaît, utilisant n'importe quelle ressource à portée de main dans les fissures et les interstices des manières de faire ordinaires, échangeant les propriétés entre matériaux inertes, matériaux animaux, matériaux symboliques, matériaux concrets et matériaux humains » (Latour, 2001, p. 201).

Sans citer Simondon (qui n'est à peu près pas traduit en anglais), certaines affirmations de Suchman sont également remarquables dans leur résonance avec la pensée de cet auteur. Suchman remarque par exemple que les réflexions féministes et celles mises de l'avant par les approches STS ouvrent sur des possibilités de considérer l'articulation de l'esprit (l'humain) et

de la matière (le corps, le non-humain, les technologies) sur des bases différentes de la conception moderne d'une intelligence autonome habitant une matière animée mécaniquement selon des « lois » de la physique. La mobilisation par Suchman du concept d'« intra-actions », repris de Barad pour appréhender les relations *à l'intérieur* des assemblages, antérieures à la démarcation sujet-objet, est également similaire à la perspective de Simondon d'une réalité préindividuelle qui constitue en elle-même une relation au monde.

Bien que la pensée de Simondon sur le processus, la continuité et la médiation trouve donc un certain écho chez les auteurs STS que nous avons cités, ces similitudes s'arrêtent cependant sur l'insistance de Simondon sur la *genèse* des objets techniques qui impliquerait l'idée d'une origine, voire d'une certaine essence des objets techniques. Ainsi, si Akrich souhaite, comme mentionné plus tôt, redonner aux dispositifs techniques leur épaisseurs, elle se distancie toutefois de l'approche « génétique » de Simondon : « Nous ne pouvons suivre Simondon dans sa genèse des objets techniques, genèse qui suppose là encore l'existence d'un moteur d'évolution intrinsèque à la technique » (Akrich, 1993, p. 91). De la même manière, Latour considère que « Simondon, pourtant, demeure classique, obsédé qu'il est par l'unité originelle et l'unité future, déduisant ses modes les uns dans les autres, d'une manière qui pourrait en fait rappeler plutôt Hegel » (Latour, 2010, p. 16). Sans nécessairement adhérer à cette idée d'une genèse des objets techniques, retenons toutefois ici cette posture épistémologique qui consiste à s'intéresser aux médiations et aux opérations par lesquels les objets se concrétisent.

Un autre horizon philosophique et épistémologique qui semble alimenter la perspective des auteurs cités dans ce chapitre est celle du pragmatisme. Déjà en 1995, Star (1995a) remarque la proximité de la théorie de l'acteur-réseau et de son principe d'inclusion des non-humains avec la préoccupation pragmatiste concernant le processus et la continuité. Pour montrer ce caractère « processuel », Star fait référence à la critique que Dewey (1896) adressait aux études psychologiques sur l'apprentissage de l'anglais, qui présentent le processus cognitif de la manière suivante : un stimulus entre dans le cerveau, s'y arrête, fait l'objet d'un processus cognitif, et ressort. Pour Dewey, au contraire, apprendre l'anglais constitue plutôt une série d'événements continuels, de changements, de réarrangements dans l'espace, le temps, et le corps (Star, 1995c). On voit bien ici les similitudes entre cette perspective de Dewey et l'importance accordée à la répétition et aux réarrangements continuels dans les travaux mentionnés plus tôt (Denis, 2006; Suchman, 2007). Star explique également les liens étroits

entre les études interactionnistes et la théorie de l'acteur-réseau par leur proximité avec les recherches pragmatistes dont le programme depuis les années 1920 est précisément de « suivre les acteurs » (Star, 1995c, p. 15). De façon plus explicite, les dernières années ont vu émerger, dans le contexte de la sociologie francophone, un nouveau style sociologique dit « pragmatique ». Selon Nachi (2006, p. 26), l'idée d'une sociologie, ou d'une science pragmatique, trouve son origine au sein de diverses disciplines telles que le pragmatisme (Peirce, James, Dewey), la sémiotique (Morris), la linguistique (Benvenis, Ducrot, Todorov) ou au sein de la théorie des actes de langage (Austin, Searle). Pour Nachi, cette épistémologie pragmatique implique une attitude que l'on pourrait qualifier d'agnostique quant à l'existence d'un « grand partage » (« the great divide ») (Nachi, 2006), ou d'une rupture, qui existerait entre deux états de savoir, deux manières « d'être-dans-le-monde » et qui prendrait notamment la forme de la division, et de la hiérarchisation, que l'on retrouve en sociologie comparative et en philosophie entre science et non-science, entre sociétés « modernes » et sociétés « prémodernes » ou encore, entre savoirs scientifiques et savoirs communs. L'idée de ce grand partage ou de cette rupture épistémologique est contestée dans la sociologie pragmatique au profit d'une perspective que l'on pourrait qualifier de *continuiste* (mais d'une continuité qui est performée), qui insiste sur le fait que les scientifiques et les acteurs partagent le même monde.

Nous retenons donc, autant chez Simondon que dans l'approche pragmatique, cette idée de continuité qui consiste à s'attarder aux médiations qui font exister les objets et les distinguent de leur environnement. En ce qui concerne les artefacts techniques, il ne s'agit donc pas de les appréhender comme des boîtes noires données d'avance, mais plutôt de nous intéresser au processus de leur démarcation et de leur stabilisation.

2.3.2 La question du relativisme et le statut du bien et de la vérité

L'accusation de relativisme est souvent adressée aux études sur les sciences et les technologies. Ces critiques, pour ne pas dire ces attaques, ont été particulièrement aiguës vers la fin des années 1990, autour de la « guerre des sciences » et en particulier à la suite de l'affaire Sokal, notamment par la publication d'un ouvrage critiquant les « impostures intellectuelles » de certains auteurs, dont Bruno Latour (Sokal et Bricmont, 1997). Réagissant à ces critiques alors encore vives, Star remarque alors que le fait de mettre en relief les

dimensions sociales de la construction des faits scientifiques vaut souvent l'accusation de relativisme et, partant de là, l'accusation de mettre sur le même pied d'égalité toutes les formes de connaissances, ou de légitimer toutes les prétentions au caractère scientifique d'une connaissance ou d'une démarche, y compris par exemple, l'idée d'une science nazie (Star, 1995b, p. 11).

Une manière de répondre à cette critique de relativisme est de considérer, comme le fait Nachi (2006), les principes de symétrie et d'impartialité, au centre des études STS, avant tout comme des principes heuristiques et méthodologiques. Il s'agit donc d'un point de vue méthodologique de ne pas distinguer a priori les connaissances scientifiques des connaissances non scientifiques, mais plutôt de s'intéresser à la façon dont les acteurs eux-mêmes distinguent ces catégories. Nachi (2006) note d'ailleurs qu'à plusieurs endroits, Callon et Latour insistent sur la scientificité de leur démarche en écrivant que le principe de symétrie est « le seul principe méthodologique qui permette d'obtenir une connaissance réaliste du processus de construction des connaissances » et qu'« il ne dit rien sur ce qu'est la raison » (Callon et Latour, 1991, p. 32). D'une certaine manière, cette perspective renvoie à ce que Hess (2001) appelle une « asymétrie de second niveau », où la méthodologie scientifique serait en fin de compte le fondement justificatif du principe de symétrie.

Un autre registre de réponse à la critique de relativisme, celle-ci plus près du pragmatisme, consiste à insister sur la stabilité de l'assemblage que constitue une « vérité » ou un « fait ». Dans *Nous n'avons jamais été modernes*, Latour (1991) distingue le relativisme absolu de sa propre posture, qu'il nomme plutôt « relativisme relativiste ». Il note que la modernité produit effectivement une science universelle, mais que cet universel est plutôt a posteriori, ou *en réseau*, dans le sens qu'une vérité n'est vérifiable partout que par l'extension des réseaux et ce, toujours localement : « Il est possible de vérifier « partout » la gravitation, mais au prix de l'extension relative des réseaux de mesure et d'interprétation » (Latour, 1991, p. 163). Star (1995a) va également dans le même sens en soutenant que l'affirmation selon laquelle la science est le résultat d'une entreprise collective qui inclut des humains et des non-humains, n'implique pas que le simple souhait d'un individu suffirait à la détruire. Affirmer que tout fait scientifique pourrait être remplacé par n'importe quelle autre proposition équivaudrait, pour Star, à une approche simpliste semblable à l'affirmation selon laquelle nous pourrions nous défaire de notre socialisation de genre : « She's been socialized as a girl. Well, let's just let her

into the corridors of power and de-socialize her » (Star, 1995a, p. 10). Dans ce sens, Star situe le danger, non pas dans l'exploration des configurations relatives à la production de connaissances, mais plutôt dans l'ignorance des conditions de cette production.

2.3.3 La participation des non-humains

Un autre aspect des travaux cités plus tôt qui a reçu beaucoup de critiques concerne le statut moral et ontologique des non-humains et de la matière. L'ensemble des travaux que nous avons cités ont en effet en commun d'accorder un rôle actif aux choses matérielles et aux non-humains. Cet aspect est souvent vu comme une tentative de mettre les humains et les non-humains sur le même pied d'égalité. En réponse à cette critique, certains chercheurs soutiennent que le principe de symétrie généralisée entre humains et non-humains, comme la symétrie restreinte entre connaissance vraie et connaissance fausse, est surtout d'ordre méthodologique et heuristique et n'a pas d'implications morales (Nachi, 2006). Nous ne sommes cependant pas de cet avis. Au contraire, certaines affirmations des auteurs cités dans ce chapitre nous semblent conférer un statut moral assez significatif aux non-humains et aux objets techniques. L'affirmation la plus « extrême » (et la plus explicite) dans ce sens est sans doute celle de Simondon qui soutient que la réintégration des objets techniques dans la culture est une œuvre comparable à celle de « l'abolition de l'esclavage et l'affirmation de la valeur de la personne humaine » (Simondon, 2001, p. 9). Latour, allant presque aussi loin que Simondon, affirme pour sa part dans son style unique et iconoclaste, ne demander « qu'une minuscule concession : qu'on étende la question de la démocratie aux non-humains » (Latour, 2004, p. 294). Clairement, nous sommes, avec ces affirmations, dans un registre traditionnellement moral et politique, qui va au-delà d'un simple principe heuristique.

Nous l'avons déjà mentionné, cette préoccupation quant au statut et au rôle des non-humains est également bien présente dans les travaux de Suchman. Elle est d'ailleurs d'autant plus importante que cette auteure a consacré sa thèse de doctorat à critiquer la conception dominante de l'humain dans le monde de l'informatique, en concluant à d'importantes asymétries entre l'humain et la machine. Dans ses travaux récents, Suchman remarque que les réflexions féministes sur les relations entre le genre et le sexe s'alignent sur les considérations mises de l'avant par les approches STS, notamment celles de considérer plus symétriquement les rapports entre humains et non-humains, ouvrant ainsi la porte à différentes

conceptualisations des rapports entre humain et matière. Suchman reconnaît ainsi les efforts de la théorie de l'acteur-réseau et de certaines théories féministes posthumanistes à appréhender la manière dont les non-humains « participent » à l'élaboration de notre monde. En revanche, elle considère important de prendre également en compte la manière dont les catégories de l'humain et du non-humain ont été historiquement constituées, et sont inégalement privilégiées. C'est dans cette perspective qu'elle note que si l'accent mis sur les non-humains peut être bénéfique dans le domaine des sciences sociales où ils sont traditionnellement marginalisés, la situation est cependant le contraire dans certains milieux où c'est plutôt l'humain qui est marginalisé au profit de la machine. De plus, elle soutient qu'il existe des asymétries humains et machines, qui s'expriment notamment dans le fait que ce sont les humains qui, au bout du compte, configurent les réseaux sociotechniques (Suchman, 2007, p. 270).

Tout en reconnaissant que les non-humains participent avec les humains, Suchman propose de faire un nouveau pas en indiquant que nous devons également reconnaître le droit des non-humains de participer de leur propre manière, qui n'est précisément pas humaine. Dans cette perspective, nous l'avons vu, Suchman rejette la métaphore de la *conversation* pour décrire notre interaction avec les artefacts communicationnels, métaphore qui suppose l'existence d'une interface parfaitement stable et transparente, qui ferait agir la machine à la manière d'un humain. Elle propose de considérer d'autres métaphores pour appréhender nos relations avec les artefacts computationnels, dans une perspective qui tiendrait compte de leur malléabilité et de leur capacité dynamique. Suivant Barad, puis Suchman, la question, qui relève autant de la politique que du design, est donc de comprendre « comment configurer les assemblages de manière à pouvoir « intra-agir » de façon responsable et générative avec eux et par eux » (Suchman, 2007, p. 285). Ainsi, comme nous l'avons précédemment mentionné, une meilleure métaphore pour décrire notre relation avec les artefacts computationnels pourrait bien être, selon Suchman, celle de l'écriture et de la lecture (Suchman, 2007, p. 23).

Suivant Suchman, et nous situant en continuité de l'esprit du logiciel libre, la question politique, plus large, qui nous préoccupe dans cette thèse consiste à savoir comment configurer les assemblages logiciels, de façon à pouvoir « intra-agir », à l'intérieur de ceux-ci. Au-delà de la question de l'accès au code source, mis de l'avant par les militants du logiciel libre, nous voulons dans le reste de cette thèse nous intéresser particulièrement aux interfaces

et aux découpages opérés *à l'intérieur* du code source, et qui impliquent différentes possibilités de participation à la conception et l'existence des technologies de l'information.

CHAPITRE III

STRATÉGIE MÉTHODOLOGIQUE

> Les recherches de type STS procèdent également par entretiens et observations et prêtent une attention particulière à la constitution et à la circulation des inscriptions (l'écrit constituant une manière répandue et efficace pour des acteurs d'agir à distance sur les situations). [...] L'analyste occupe une position d'observation privilégiée et nettement séparée de celle des agents impliqués dans les systèmes collectifs qu'il étudie (Licoppe, 2008, p. 295).

Ce chapitre présente la démarche méthodologique de notre enquête de terrain dont les résultats sont présentés dans les trois chapitres suivants. L'enquête prend pour terrains *symfony* et *SPIP*, deux logiciels développés publiquement et collectivement sur Internet et orientés vers la conception de sites web interactifs. Elle s'appuie sur une vingtaine d'entrevues ainsi que sur l'analyse des discussions, controverses et négociations des acteurs dans leur activité collective d'écriture du code source. La première partie du chapitre tente de situer notre approche face aux questionnements contemporains concernant la démarche ethnographique et en particulier, l'ethnographie dite « multisite ». La deuxième partie présente nos deux terrains, en insistant sur les traits communs et divergents entre ceux-ci. La troisième partie présente en détail les techniques d'enquêtes qui ont été mobilisées, ainsi qu'une réflexion sur les aspects éthiques de la recherche. La quatrième partie précise quelques notes concernant nos méthodes et notre analyse.

3.1 Approche méthodologique

L'objectif de cette première partie est de lier les réflexions théoriques que nous avons présentées plus tôt à nos choix méthodologiques concrets. Nous explorerons certaines approches s'identifiant à la démarche ethnographique, dans le contexte des études STS. Nous verrons que pour plusieurs auteurs que nous aborderons, l'ethnographie revêt une définition assez large dont les contours sont parfois assez éloignés de la définition traditionnelle de l'ethnographie, au point même où certains auteurs, tels que Hine (2007) se questionnent sur la

pertinence même de qualifier leurs approches d'ethnographiques. Nous inspirant de travaux de cette dernière auteure, nous souhaitons appréhender ici la tradition ethnographique, non pas comme méthode stricte, mais plutôt comme un guide et une source d'inspiration.

3.1.1 Introduction : l'ethnographie dans les études STS

Latzko-Toth (2010) remarque que la plupart des études en STS s'identifient à une démarche ethnographique, bien que plusieurs, et en particulier les études s'inscrivant dans la théorie de l'acteur-réseau, ont progressivement délaissé l'observation directe, pourtant centrale à la démarche ethnographique traditionnelle. Latzko-Toth remarque par exemple que Bruno Latour, dans son étude de cas d'Aramis (Latour, 1992a), s'identifie comme un ethnographe alors qu'il a commencé son enquête au moment où Aramis était sur le point d'être abandonné. La méthodologie dans ce cas était beaucoup plus près de l'enquête historique que de l'observation directe, et a ainsi consisté notamment « à fouiller pour retrouver les restes de prototypes, de voies [ferrées], de documents, tout comme le technologue [étudiant] des techniques traditionnelles perdues dans la nuit des temps » (Latour, 1992a, p. 386). L'ambiguïté du caractère ethnographique des études STS – et en particulier celles conduites par Bruno Latour – a également été remarquée par Patrice Flichy qui note que si Latour, au début de son étude sur la vie de laboratoire, présente son étude comme ethnographique, cette perspective ethnographique disparaît par la suite :

> Dans *La vie de laboratoire*, l'observation de l'activité scientifique au quotidien ne constitue qu'une partie de l'ouvrage. Latour y présente notamment une remarquable analyse des conversations informelles de laboratoire, des hésitations des scientifiques face à des phénomènes inattendus, du rôle du hasard dans l'élaboration des solutions. Pour le reste, il utilise la panoplie classique de l'historien ou du sociologue qui effectue des études monographiques : analyse systématique des documents écrits (articles, comptes-rendus de réunion), interviews (Flichy, 1995, p. 107).

Hess (2001) note pour sa part une certaine prise de distance par rapport aux définitions canoniques de l'ethnographie dans les études STS. Pour Hess, les premières études STS étaient surtout concernées par l'ethnographie de laboratoire et l'approche était surtout d'ordre microsociologique, s'appuyait largement sur l'observation et concernait un terrain bien

délimité[50]. Les études STS ont progressivement abordé des terrains aux contours parfois moins bien définis en s'appuyant davantage sur des sources documentaires et des entrevues que sur l'observation directe. Ces transformations seraient d'abord significatives du « tournant technologique » qui a marqué ces études au début des années 1990, tournant dans lequel la théorie de l'acteur-réseau s'inscrit notamment. Ensuite, elles seraient le fait d'une plus grande préoccupation dans les études STS pour l'analyse des différentes manières dont la société et la connaissance se façonnent mutuellement, en quittant les laboratoires pour plutôt s'intéresser aux points de vue des groupes de profanes, des activistes, des mouvements sociaux et de la culture populaire. Citant Marcus (1995), Hess (2001) note que les études dans cette seconde génération d'études STS tendent, de façon explicite ou implicite, à adopter une démarche plus « multisituée ». Hess recense les méthodes ethnographiques « multisituées », utilisées dans ces recherches STS de deuxième génération :

> Assister à des congrès (un site préféré aux laboratoires par les membres de la deuxième génération), travailler dans des laboratoires et des maisons d'enseignement, assister à des conférences virtuelles et présentielles, interviewer un large spectre de personnes associées à la communauté, lire une vaste littérature technique, faire du travail d'archive, développer des relations à long terme avec des informateurs (qui peuvent, avec le temps, devenir des amis ou même des cochercheurs), interviewer des *outsiders* et des individus ordinaires sur leur perception de la communauté d'experts et de ses extrants (*products*), devenir membre d'organisations activistes et de mouvements sociaux, et offrir des services et de l'aide à la communauté (par exemple, en rédigeant des textes ou en présentant des conférences sur les aspects sociaux, historiques et réglementaires (*policy*) de la communauté) (Hess, 2001, p. 239)[51].

Hine, qui avait auparavant publié quelques travaux sur l'« ethnographie virtuelle » (Hine, 2000; Hine, 2005)[52], soutient également dans un article plus récent, l'intérêt de l'utilisation de l'ethnographie multisite comme méthodologie de « moyenne portée » (« middle-range ») dans les études contemporaines en STS. Elle s'inspire en cela de l'idée d'une théorie de moyenne

---

50 Évidemment, comme mentionné quelques lignes plus tôt, certaines nuances doivent être apportées concernant le caractère effectivement ethnographique de ces premières études.

51 Nous avons ici utilisé la traduction réalisée par Latzko-Toth (2010, p. 84).

52 Notons que l'approche des *Virtual Methods,* notamment développée par Hine, est aujourd'hui contestée par l'approche des *Digital Methods.* Dans un ouvrage soutenant cette dernière approche, Rodgers *(2009)* soutient la nécessité de développer des méthodologies « nativement numérique », en opposition aux *Virtual Methods* qu'il voit davantage comme une importation des approches traditionnelles pour analyser la réalité numérique.

portée (*middle-range theory*) par le sociologue des sciences Robert Merton (1973). Dans une étude sur le développement et l'utilisation des technologies de l'information dans les sciences biologiques, elle note qu'en étudiant comment ces développements font sens pour les acteurs, elle a été amenée à poursuivre son étude dans différents sites, comme des forums de discussions, des musées, des jardins botaniques, des documents politiques, des sites web, des journaux, des conférences, des entrevues, des courriels et des conversations informelles. Pour Hine, cette recherche pourrait être qualifiée de « multisite » dans le sens qu'elle s'est attelée à l'étude de différents sites pour étudier son phénomène. Selon elle, la force des approches multisites concerne leur volonté d'établir des connexions plutôt que d'accepter des frontières du terrain, ce qui peut sembler évident à première vue.

La notion d'ethnographie multisite a d'abord été mise de l'avant par Marcus (1995). L'ethnographie multisite postule que la conception de territoire fermé constitue une façon artificielle de construire des objets de recherche et que les terrains ont toujours été plus ou moins construits et délimités par les ethnographes. De plus, l'ethnographie multisite reconnaît que les sociétés contemporaines sont de plus en plus caractérisées par la mobilité, les connexions et la communication. Dans cette optique, il est de plus en plus difficile d'analyser à partir d'un terrain circonscrit. L'approche d'ethnographie multisite s'articule plutôt autour de chaînes, de chemins et de juxtapositions et est orientée vers une logique d'association ou de connexion (Marcus, 1995, p. 105).

Parmi les modes de construction d'une ethnographie multisite, Marcus propose notamment de suivre la circulation d'un objet (*follow a thing*), à travers les différents contextes dans lequel celui-ci se manifeste. Marcus cite l'introduction de la collection The *Social Life of Things*, rédigé par Appadurai (1988), qui constituerait selon lui, l'expression la plus importante de l'approche « suivre une chose », en retraçant les différents statuts des choses dans différents contextes :

> In tracing the shifting status of things as commodities, gifts, and ressources in their circulations through different contexts, Appadurai presumes very little about the governance of a controlling narrative of macroprocess in capitalist political economy but allows the sense of system to emerge ethnographically and speculatively by following paths of circulation (Marcus, 1995, p. 107).

Marcus note que les travaux de Latour et d'Haraway ont joué un rôle crucial dans le développement de l'ethnographie multisite dans les études sur les sciences et les technologies, et en particulier, dans le développement d'une approche multisite consistant à « suivre une chose ». Marcus considère que Haraway va plus loin que Latour en s'intéressant autant au sens matériel, mais également métaphorique de la chose : « Latour's work exemplifies this mode, albeit less so than does Haraway's which has as much a metaphorical as a material sense of the things she traces » (Marcus, 1995, p. 108). Marcus accorde d'ailleurs dans cet article un place importante à l'œuvre d'Haraway.

L'approche consistant à « suivre une métaphore » constitue pour Marcus un autre type d'approche multisite : « This mode involves trying to trace the social correlates and groundings of associations that are more clearly alive in language use and print or visual media » (Marcus, 1995, p. 108). Pour Marcus, cette approche serait le mode privilégiée de construction de l'objet de recherche dans l'œuvre d'Haraway. En anthropologie, il considère l'ouvrage de Martin (1994) comme l'un des meilleurs exemples de l'approche consistant à « suivre la métaphore ». Dans cette étude, l'auteure s'intéressait d'abord aux différentes manières de penser le système immunitaire humain dans différents contextes. Elle en est finalement venue à s'intéresser à la métaphore de la « flexibilité », très présente dans les conceptions scientifiques du système immunitaire, et à établir des articulations avec l'usage de cette même métaphore , dans d'autres contextes tels que la théorie de la complexité, les théories et les pratiques du management, et finalement, dans les nouvelles idéologies du travail.

D'une certaine manière, cette approche consistant à « suivre une métaphore » s'apparente à celle utilisée par Serge Proulx (2007c) dans un article où il retrace les différents usages de la métaphore de la « société de l'information ». Bien que l'étude en question ne relève pas explicitement de l'ethnographie, l'auteur propose néanmoins une approche qui « oriente l'attention de l'observateur vers les métaphores utilisées dans le discours des sujets ». Dans cet article, Proulx réfère à la théorie constructiviste des métaphores de Krippendorf, qui considère que les métaphores organisent les expériences de communication des acteurs, et peuvent par conséquent participer à créer des réalités (Krippendorff, 1993, p. 5). Faisant référence à Krippendorf, Proulx explique également que le choix d'une métaphore peut être susceptible de devenir l'objet de controverses :

> Toute métaphore, tout terme employé par des acteurs sociaux pour décrire la réalité existante ou perçue comme émergente – ou pour tracer le projet collectif auquel ces acteurs aspirent – est susceptible de devenir l'objet de controverses sociales et politiques. Ces controverses portent sur les significations mêmes des termes, engendrant ainsi dans l'espace public, des débats sociaux autour des définitions, des luttes sociales et sémiotiques de classification (Proulx, 2007c, p. 113).

Suivant cette approche, l'auteur analyse ainsi les discours des acteurs qui font usage de la métaphore de la société de l'information depuis la décennie 1970.

3.1.2 Déplier l'objet technique, laisser parler les acteurs

Notre approche pourrait se résumer à *déplier le code source* et à *laisser parler les acteurs*. Dans un article que nous avons cité au chapitre 2 (p. 65), sur la performativité des artefacts, Latour (2010) propose de parler de *pliage technique* pour désigner des réalités telles que « le montage si délicat d'habitudes musculaires qui font de nous, par apprentissage, des êtres compétents doués d'un fin savoir-faire », ou encore, beaucoup plus près de notre préoccupation, « pour désigner la distinction entre un logiciel et son compilateur » (Latour, 2010, p. 30). La notion de pli a également été mise de l'avant dans un autre article de cet auteur, qui critique la distinction entre moyen et fin pour analyser les technologies. Dans cet article, Latour (2000) insiste sur le fait que les objets techniques ne sont jamais complètement contemporains de nos actions, mais conservent plutôt des temporalités hétérogènes repliées les unes sur les autres. Utiliser un marteau par exemple, c'est mettre sa main sur de la matière aussi vieille que la planète et reproduire des techniques millénaires qui sont réarticulées à chaque époque. D'un point de vue méthodologique autant que théorique, il s'agit donc de passer d'une conception d'acteurs humains reliés par un médium instrumental, à une conception de la médiation où tout est un réseau d'associations plus ou moins stabilisées entre humains et non-humains. Dans cette perspective, l'objet technique n'est pas appréhendé comme une boîte noire « jetée » dans la société, mais plutôt comme un assemblage hétérogène articulé dans un réseau et progressivement solidifié par le travail des acteurs qui s'y impliquent.

La démarche méthodologique de la théorie de l'acteur-réseau, qui se veut surtout ethnographique, consiste à retracer ces médiations qui font exister les objets culturels, techniques, scientifiques ou sociaux et de documenter les controverses qui participent à la

solidification de ces médiations. La notion de controverse est ici centrale, car c'est dans le cadre des controverses que l'on peut le mieux retracer ces médiations, et suivre les acteurs dans la composition de ces objets : « Ces controverses fournissent à l'analyste une ressource essentielle pour rendre traçables les connexions sociales » (Latour, 2006, p. 46). Il s'agit donc de suivre les acteurs dans ces controverses et de s'appuyer, pour l'analyse, sur les catégories que les acteurs eux-mêmes mobilisent : « La tâche de définition et de mise en ordre du social doit être laissée aux acteurs eux-mêmes, au lieu d'être accaparée par l'enquêteur » (Latour, 2006, p. 26). Dans le cadre d'une analyse du code source, il s'agit de suivre les différentes controverses portant sur la réalisation, la révision et la délimitation du code source.

Notons finalement pour terminer la proposition ethnographique de Suchman d'un engagement « in multiple, partial, unfolding, and differentially powerful narratives » (Suchman, 1999; Suchman, 2008), qui semble également relevé d'un caractère « multisite ». Suchman insiste également sur le fait que les recherches portant sur des objets sociomatériels doivent s'attarder particulièrement à deux aspects : le travail de démarcation et de découpage du réseau par lequel les entités sont délimitées en tant que telles, ainsi que la localisation de ces entités dans le contexte de relations spatiales et temporelles (Suchman, 2007, p. 283).

3.1.3 Le travail invisible : ethnographie des infrastructures

Dans son article intitulé « The Ethnography of Infrastructure », Susan Leigh Star (1999) propose une démarche d'analyse des infrastructures d'information qu'elle qualifie d'ethnographique. L'infrastructure, chez cette auteure, se définit tout d'abord par son invisibilité, par le fait qu'elle se situe toujours en arrière-plan d'autres types d'activités. L'infrastructure, dans cette perspective, est donc fondamentalement relationnelle puisqu'un artefact peut être considéré comme une infrastructure pour certains, alors qu'il est l'objet même du travail pour d'autres. Dans notre cas, par exemple, le code source constitue une infrastructure qui soutient l'usage du web participatif, tandis qu'il est l'objet même de l'activité pour les programmeurs.

Sans en porter le nom, cette approche d'une « ethnographie des infrastructures » pourrait être considérée comme relevant d'une telle approche « multisite ». Star (1999) note que l'analyse qualitative des infrastructures de communication qui sont utilisées par des centaines, voire des milliers de personnes, pose des problèmes importants à une approche qui se voudrait

qualitative et ethnographique, mais qui est traditionnellement plus adaptée à des terrains plus circonscrits. L'auteure note également que malgré la prolifération de traces en ligne et de « notes de terrain » prêtes à l'usage dans le cas des infrastructures d'information, la dispersion du matériel reste un véritable défi pour en arriver à une analyse cohérente, d'ordre qualitatif. Malgré cette dispersion et cette délimitation ambiguë du « terrain », l'auteure insiste toutefois sur la pertinence d'un outil de type ethnographique et qualitatif, pour sa capacité à saisir les voix silencieuses et les différents sens : « Its strength has been that it is capable of surfacing silenced voices, juggling disparate meanings, and understanding the gap between words and deeds » (Star, 1999, p. 383).

Star préconise ainsi d'intégrer une « sensibilité ethnographique » à l'analyse des infrastructures, c'est-à-dire de garder à l'esprit que les gens ont certaines représentations de leur situation. L'auteure propose ainsi deux manières d'analyser les infrastructures technologiques (Star, 1999). D'une part, il s'agit d'identifier un ou des « récits maîtres » (« *master narratives* ») qui seraient inscrits ou matérialisés dans les infrastructures, et de faire ressortir certaines de leurs expressions. Star fait par exemple référence au travail de Latour (1992a) qui fait ressortir la manière dont le système de métro Aramis « encode » une grandeur particulière de voiture basée sur la famille nucléaire. Elle donne également l'exemple des bandages et des prothèses de « couleur peau » qui expriment en fait seulement la peau « blanche » et rend ainsi invisible la diversité des tons de peau. D'autre part, l'auteure préconise d'intégrer une sensibilité ethnographique vers le travail qui n'est pas formellement reconnu. L'ethnographie des infrastructures, pour Star, consiste à performer ce que Bowker (1994) appelle un « renversement infrastructurel » (*infrastructural inversion*) pour mettre en avant-plan les éléments souvent moins visibles du travail (Star, 1999, p. 380). L'auteure propose par exemple d'examiner plus précisément les moments de discussions et de décisions concernant l'encodage et la standardisation, et les activités de bricolages. Par exemple, les concierges, les cuisiniers, les gardiens de sécurité et les secrétaires ne sont à peu près jamais reconnus dans le cadre d'une publication scientifique. Star (1999) note par ailleurs que certains types de travail, voire certaines conventions, demeurent davantage tacites que formalisés, en partie pour échapper à un contrôle externe. Dans une de leurs études, Star et Ruhleder (1996) avaient par exemple rendu compte de moments importants dans la carrière d'un biologiste – particulièrement au niveau du postdoctorat – où le secret est davantage

valorisé que le partage des résultats préliminaires dans des lieux semi-formels (Star, 1999, p. 385).

Finalement, l'auteure insiste sur la nécessité d'analyser les standards ou les classifications formels (Star, 1999, p. 379). Elle appelle également à analyser les débats, ou les controverses, concernant les noms de domaines, les protocoles ou le choix des langages de programmation. Pour Star, l'un des aspects les plus importants dans l'étude des infrastructures concerne l'inscription de valeurs et de principes éthiques dans les profondeurs de l'environnement informationnel. Sur le plan méthodologique, il s'agit donc d'enquêter sur les traces visibles laissées par les codeurs, les designers et les usagers d'un système informatique, mais également d'aller en « arrière-scène » pour mettre en évidence le travail invisible.

3.1.4 Anxiétés méthodologiques : aux limites de l'ethnographie

Plusieurs auteurs ont toutefois soulevé certains questionnements, voire certaines « anxiétés méthodologiques[53]» (Marcus, 1995, p. 99), par rapport à l'approche de l'ethnographie dite « multisite », en regard des définitions traditionnelles de l'ethnographie. Hine (2007) note toutefois que l'ethnographie « multisite », tout en s'efforçant de décrire adéquatement le phénomène étudié, peut parfois ne pas être considérée comme une ethnographie, au sens canonique du terme. L'auteure, qui avait quelques années auparavant publié un ouvrage sur l'« ethnographie virtuelle », explique sa sensibilité grandissante envers l'idée d'utiliser la catégorie de l'ethnographie. Dans cet article, Hine note que l'usage de « l'ethnographie virtuelle » était un jeu de mots, dont l'objectif était d'insister sur une relation ambivalente avec les pratiques et les principes canoniques de l'ethnographie, mais en même temps, exprimait un désir de rester en dialogue avec ces traditions. L'ethnographie virtuelle permettait d'insister sur l'observation des pratiques en ligne, dans un esprit ethnographique, mais également des pratiques hors ligne. En d'autres termes, il ne s'agit pas tant dans cette perspective d'adopter les canons de l'ethnographie comme une fin en soi, mais plutôt de s'inspirer de ces méthodes pour développer une approche permettant de rendre compte adéquatement de la réalité étudiée, et de la partager avec différents publics. La tradition

53 Nous retenons ici ce terme d'« anxiété méthodologique », qui est le titre d'une section de l'article de Marcus (1995), car c'est de cette façon que nous avons vécu notre relation avec la démarche ethnographique durant cette recherche.

ethnographique, dans cette perspective, demeure importante, mais agit surtout comme guide et source d'inspiration, plutôt que comme méthode stricte. Hine remarque ainsi une ambivalence similaire face à l'ethnographie dans d'autres travaux, par exemple, ceux de Jensen (2004) qui qualifie sa démarche de « quasi-ethnographique » ou encore ceux de Star (1999), qui propose d'intégrer une « sensibilité ethnographique » à sa démarche.

Latzko-Toth (2010) remarque pour sa part que le caractère « ethnographique » des études de cas en STS a en premier lieu lié à l'injonction de « description dense », mise de l'avant dans l'anthropologie de Clifford Geertz (1998). Le terme ethnographie, dans ce cas, renvoie davantage à l'idée de descriptions denses, locales, en contexte d'un phénomène, plutôt que par l'observation directe et l'immersion dans un terrain donné[54]. Latzko-Toth fait également référence à Hammersley et Gomm (2000) qui donnent – selon Latzko-Toth – une « saveur » (les guillemets sont de l'auteur) plus ethnographique à l'étude de cas, en attribuant beaucoup plus d'importance au sens que les acteurs donnent à leurs conduites, qu'à l'analyse externe du chercheur. Pour ces auteurs, le rôle du chercheur consisterait davantage à « donner une voix aux acteurs qu'à les utiliser comme des informateurs ou des répondants » (Hammersley et Gomm, 2000, p. 3; cité par Latzko-Toth, 2010, p. 79).

C'est cette dernière perspective, « donner une voix aux acteurs », qui retient particulièrement notre attention dans notre propre démarche. La différence principale entre l'approche développée dans ce chapitre et une approche ethnographique tient probablement de la « mise en récit », et en particulier, de sa variation dans le temps. Bien que certains moments de nos descriptions sont plus ethnographiques et insistent plus particulièrement sur cette variation dans le temps, notre objectif est surtout dans ce cas de nous attarder aux points de vue des acteurs.

---

54 Latzko-Toth (2010) note toutefois un glissement, dans les études ANT, vers une tendance à considérer le cas comme un prétexte ou un support à la théorisation – ce qu'il appelle une étude de cas illustrative – plutôt que comme une localité d'où il est possible d'étudier un phénomène donné, dans une démarche propre de la théorisation ancrée. Latzko-Toth parle également d'« étude de cas imaginaire » pour décrire l'approche de certains travaux ANT. L'exemple paradigmatique des portes et des ferme-porte de La Villette de Latour (1992b) est exemplaire de cette approche de l'étude de cas. Elle a d'ailleurs été critiquée par Collins et Yearley (1992).

3.2 Terrains : deux « projets PHP »

> A priori, il y a environ six millions de développeurs PHP dans le monde (sf02).

Comme mentionné en problématique, nos analyses s'attardent plus précisément à deux logiciels écrits dans le langage informatique PHP et destinés au fonctionnement de sites web. Ces deux logiciels sont SPIP et symfony.

Le choix d'étudier ces deux logiciels était principalement motivé par le fait que l'étude a été réalisée dans le cadre d'une entente de cotutelle France-Québec. Au moment de choisir ces projets, l'un de nos objectifs était en effet de circonscrire temporellement et spatialement notre enquête afin de profiter de notre séjour dans ce pays. Nous avons ainsi choisi ces projets car leurs dirigeants et la plupart de leurs contributeurs habitent en France. Comme nous le verrons plus loin, ces deux projets sont par ailleurs intéressants à analyser étant donné leurs différences aux niveaux des valeurs et du mode d'organisation. Les deux sections suivantes décrivent plus en détail les projets étudiés, tandis que la troisième se termine par une réflexion sur ce qui les distingue et les réunit. Le tableau 3.1 présente un portrait d'ensemble de ce qui caractérise ces deux projets :

Tableau 3.1 : Portrait des deux projets étudiés

SPIP	symfony
Système de gestion de contenu (CMS) Lancement en 2001	**Cadre d'application – *Framework* – web** Lancement en 2005
Quelques chiffres : Liste [spip-dev] : 4 089 courriels/111 pers. Liste [spip] : 5 316 courriels/331 pers. Taille du code (coeur) : 14,015 Mb	**Quelques chiffres :** Liste [symfony-dev] : 1 675 courriels/150 pers. Liste [symfony-users] : 11 861 courriels/1 706 pers. Tailles du code (coeur) : 14,334 Mb
Sites notables : Le Monde Diplomatique; La Poste	**Sites notables :** Delicious; Yahoo bookmarks; Daily Motion
Orientation militante/associative, française; Non institutionnalisé	Orientation commerciale, internationale; soutenu par une entreprise

Note : Ces chiffres, calculés à partir des courriels envoyés entre le 1er juillet 2009 et 29 juin 2010, sont présentés à titre indicatif et ne visent qu'à donner une image générale de l'envergure de chacune des communautés. D'autres techniques utilisées pour réaliser ces calculs ont donné des résultats différents, bien que toujours dans ces ordres de grandeur. Une véritable analyse quantitative devrait discuter davantage des méthodes utilisées pour obtenir ces résultats.

3.2.1 Le code source de SPIP

SPIP, c'est pas une communauté d'informaticiens (spip09).

Le logiciel SPIP est décrit comme « un système de publication pour l'Internet qui s'attache particulièrement au fonctionnement collectif, au multilinguisme et à la facilité d'emploi[55]». C'est un logiciel libre, distribué sous la licence GNU/GPL. Ce logiciel est né en 2001 de l'initiative d'un collectif défendant le web indépendant et la liberté d'expression sur Internet[56], mais selon la mythologie « spipienne », les débuts de SPIP remonteraient à l'année 1998 lorsque plusieurs des acteurs précurseurs aux prises avec des besoins de création de sites web se sont rencontrés pour tenter de mettre en commun leurs forces (SPIP, 2002). Wikipédia décrit ainsi la signification de l'acronyme SPIP :

> SPIP est un « acronyme signifiant « Système de publication pour l'Internet » ; le dernier « P » est laissé à la libre interprétation de chacun et est souvent traduit par « partagé » ou « participatif », dans la mesure où ce logiciel permet surtout d'éditer collectivement un site[57].

L'acronyme SPIP renvoie cependant aussi au personnage de l'écureuil dans la populaire bande dessinée Spirou, dont le nom était également SPIP. C'est ce qui explique également la signification du logo de SPIP (voir tableau 3.1). Nous verrons dans les chapitres suivants que l'« imaginaire de l'écureuil » a également marqué plusieurs choix en termes de nomenclatures (voir par exemple la section 4.1.5 à propos du terme de « noisette » qui désigne des petits morceaux de code source pouvant être réutilisés dans la fabrication d'un site web).

Aujourd'hui, des dizaines de milliers de sites web utilisent ce logiciel pour des raisons très diverses (militantes, commerciales, gouvernementales, différents contextes culturels et linguistiques)[58]. La communauté de SPIP est assez active et principalement française. Le code de SPIP est rédigé en français, si on fait exception des mots empruntés au langage PHP[59].

---

55 <http://www.spip.net/rubrique91.html> (consulté le 23 novembre 2011).

56 <http://www.spip.net/rubrique91.html> (consulté le 6 décembre 2011).

57 <http://fr.wikipedia.org/w/index.php?title=Système_de_publication_pour_l'Internet&oldid=62076335> (consulté le 26 février 2012).

58 Un courriel sur la liste spip-dev (19 avril 2010) indiquait qu'il y avait 11 604 sites web recensés, fonctionnant sous SPIP.

59 Voir également l'historique de SPIP sur Wikipédia :

Bien qu'à notre sens le projet semble encore très vivant, il a souvent été critiqué par des acteurs d'autres projets (dont certains que nous avons rencontrés en entrevue) pour être un logiciel « dépassé » (Libre-Fan, 2008) Le terme « dépassé » est évidemment contesté par les acteurs de SPIP. Cependant, la plupart des acteurs que nous avons rencontrés, autant dans SPIP que dans symfony, seraient d'accord pour dire que SPIP contraste avec plusieurs autres projets PHP. À tous le moins, nous pourrions dire ici que ces projets sont animés par des valeurs et des pratiques différentes, distinctions que nous mettrons en évidence tout au long de la thèse, à commencer par la section 3.2.3, un peu plus loin.

3.2.2 Le code source de symfony

Le logiciel symfony est décrit comme un « full-stack framework, a library of cohesive classes written in PHP5 [60]». Au contraire d'un logiciel aux contours bien définis, symfony doit plutôt être appréhendé comme un ensemble de composantes logicielles distinctes. Le développement de symfony s'appuie sur une communauté internationale très active, dont la base est française. Contrairement à SPIP, le code de symfony est uniquement rédigé en anglais, de même que l'ensemble de la documentation et des courriels de discussion du développement. La page « About » du site symfony-project.org insiste sur la propreté du design et la lisibilité du code source (« clean design and code readability »). Les prochains chapitres de notre thèse reviendront sur la manière dont ces « principes » de développement de logiciel marquent l'identité du projet.

3.2.3 Premiers constats : distinctions et similitudes entre les projets étudiés

L'objectif de notre étude n'est pas réaliser une étude de cas détaillé de ces projets, mais plutôt de les prendre comme « terrains ». Il est difficile d'établir un profil sociologique précis pour chacun des projets étudiés, à partir de notre approche de recherche, alors nous nous contenterons ici de présenter certaines distinctions et similitudes précises, de façon à leur donner une certaine épaisseur. Ces distinctions sont particulièrement marquées au niveau des valeurs qui animent chacun des projets.

<http://fr.wikipedia.org/wiki/Syst%C3%A8me_de_publication_pour_l%27Internet#Historique> (consulté le 26 février 2012).

60 <http://www.symfony-project.org> (consulté le 1er novembre 2011).

Une première distinction entre ces projets concerne leur nature même. Plusieurs acteurs et auteurs, incluant le site Wikipédia, définissent en fait SPIP comme un *Content Management System* – « CMS » –, soit un logiciel aux contours bien définis, tandis que symfony serait plutôt un « Framework », c'est-à-dire un ensemble de « briques logicielles », ou de morceaux de code source, qu'il faut ensuite rassembler. Cette distinction n'est pas à négliger, puisque quelques acteurs (ou personnes périphériques) à qui nous avons présenté les deux projets à l'étude considèrent que ceux-ci sont assez différents, voire qu'ils ne peuvent pas être comparés. Nous croyons cependant que cette distinction doit être nuancée. En effet, s'il a été initialement conçu comme un CMS, l'organisation du code source SPIP est, de l'avis de plusieurs acteurs, de plus en plus orientée vers le « paradigme Framework ». De manière symétrique, dans symfony, la proposition de créer un CMS basé sur symfony a été soulevée au moins une fois lors de la conférence *Symfony Live 2010*. Dans un sens, nous pourrions dire que les deux technologies se rejoignent dans la mesure où symfony cherche notamment à se particulariser comme un CMS, tandis que SPIP se généralise en Framework[61].

Un deuxième ensemble de distinctions, plus sociologique, concerne l'origine nationale et professionnelle des acteurs qui s'y impliquent. SPIP étant un projet surtout francophone, il rejoint difficilement des acteurs situés à l'extérieur de la France[62]. Symfony, au contraire, se démarque par la participation d'acteurs de divers pays tels que la France, les États-Unis, l'Allemagne et les Pays-Bas. Dans les deux cas, c'est surtout une présence « blanche », occidentale que l'on remarque, bien que dans le cadre d'une conférence de symfony, nous avons pu remarquer la présence de participants d'origine ou de citoyenneté probablement Indienne. Une autre grande différence concerne la « filière » professionnelle d'où proviennent les acteurs des projets. Dans symfony, la plupart des personnes interrogées sont détentrices d'un diplôme de premier cycle en informatique, en ingénierie ou en multimédia, dans le cas d'une actrice (femme) que nous avons rencontrée en entrevue. Dans le cadre du projet SPIP, on peut constater un éclectisme des origines professionnelles des acteurs. Alors que certains des acteurs que nous avons rencontrés ont terminé des études de troisième cycle (un des

---

61 Comme mentionné plus loin, ces deux logiciels ont également comme compétiteur commun le logiciel *Drupal*, ce qui montre bien que, malgré des différences importantes, SPIP et symfony restent quand même comparables sous plusieurs points.

62 Au Québec, le projet SPIP n'a pas non plus réussi à percer, ce qui nous a d'ailleurs valu plusieurs questionnements et insistances pour justifier le projet SPIP.

acteurs de SPIP est même professeur titulaire), d'autres sont autodidactes et n'ont souvent reçu aucune formation professionnelle en informatique, ou aucune formation universitaire. Ainsi, parmi les personnes que nous avons rencontrées, celles-ci avaient par exemple étudié en sociologie, en théologie, en mathématique, en littérature classique, en génie industriel ou en informatique théorique. Finalement, en ce qui concerne la participation des femmes, on peut constater une participation nettement plus grande de celles-ci au sein de SPIP qu'au sein de symfony. Ainsi, dans le cadre de la première conférence *Symfony Live* à Paris, à laquelle nous avons participé, nous avons constaté la présence de 2 ou 3 femmes seulement, sur un auditoire de 200 à 300 personnes. Dans un SPIP-Party, au contraire, une dizaine de femmes étaient présentes, sur une cinquantaine de personnes. Cette question est d'ailleurs explicitement discutée au sein de SPIP, ce que nous n'avons pas remarqué dans symfony[63]

Finalement, un dernier ensemble de distinctions concerne les valeurs qui animent chacun des projets. Ces différences, que nous approfondirons à différents moments dans les prochains chapitres, apparaissent à plusieurs niveaux pour chacun des projets. Disons à ce point qu'il est assez aisé de remarquer une tendance nettement plus militante, voire anarchisante, du projet SPIP, comparée à celle, plus commerciale, du projet symfony. L'une des manières de constater cette différence est sans doute dans la manière dont les conférences publiques sont organisées. Durant la période de l'enquête, nous avons par exemple participé à deux conférences publiques, pour chacun des projets. Dans le cadre de symfony, ces rencontres se déroulent sous la forme d'une conférence au coût de 200 euros, et sont commanditées par plusieurs entreprises informatiques telles que Microsoft ou Yahoo. Les conférence réunissent plusieurs centaines de personnes, dans une grande salle ou un auditorium de la Cité Internationale Universitaire de Paris. Dans le cas de SPIP, les *SPIP-Party,* qui tiennent lieu de conférence annuelle, sont au contraire gratuits, et sont organisés sur une base collaborative et beaucoup plus collégiale. Chacun y apporte sa nourriture et partage les mets de son pays d'origine. Les rencontres sont organisées quelques jours seulement à l'avance, voire la journée même. Ces rencontres ne réunissent que quelques dizaines de personnes.

---

63 Dans la deuxième conférence à laquelle nous avons participé à propos symfony, une des participantes avait toutefois félicité le présentateur dans son fil Twitter pour avoir utilisé le personnage d'une femme dans sa présentation numérique.

Ces différences en termes de valeurs s'expriment notamment dans le type de licence privilégié par chacun des projets. En effet, alors que la licence publique générale GNU (GPL) est la licence privilégiée pour SPIP, son utilisation est interdite dans le cas de symfony. En d'autres termes, le code source de symfony est légalement incompatible avec celui de SPIP et il est par conséquent impossible (légalement) de partager des morceaux de code source entre les deux projets. L'extrait suivant montre bien les appréhensions du « sponsor » du projet vis-à-vis les licences de type GPL :

> I (as in the symfony project sponsor) only want to host (package files and subversion repositories) plugins released under a MIT/BSD/LGPL/... license on the symfony-project.com website. People can release plugins under other licenses (GPL or whatever) if they want but I don't want to host them[64].

Cette constatation, que nous avons remarquée assez tardivement dans notre étude, nous a surpris. L'analyse du site web de symfony ou de ses listes de discussions ne permettent pas de faire aisément ressortir les raisons de ce choix. Toutefois, d'autres sites web faisant la promotion de symfony indiquent que le choix de symfony d'utiliser des licences moins restrictives que la licence GPL, par exemple la licence MIT[65], permet la réutilisation du code source de symfony dans le développement d'applications propriétaires. Cette possibilité est plutôt interdite par les licences GPL, qui sont les seules autorisées par SPIP, empêchant ainsi l'utilisation de SPIP dans le cadre du développement de logiciels propriétaires. Comme nous le montrerons dans les chapitres suivants, l'un des traits qui ressort de notre étude concerne ce que l'on pourrait appeler cette « inscription » (Akrich, 1993) des valeurs de chacun des projets dans leur code source. Nous remarquerons par exemple certaines relations entre la forme du code de SPIP et une certaine diversité sociale au sein de ce projet.

3.3 Collecte de données et méthodes d'enquête

Cette section présente les différentes méthodes et techniques qui ont été mobilisées pour former et rassembler notre corpus de données. Les trois premières sections présentent ces méthodes, soit l'observation présentielle (section 3.3.1), la réalisation d'entrevues semi-

---

64 <http://www.mail-archive.com/symfony-users@googlegroups.com/msg02883.html> (consulté le 1er novembre 2011).

65 Comme son nom l'indique, la licence MIT a été créée et publiée par le Massachusetts Institute of Technology.

dirigées (3.3.2) et l'analyse des traces en ligne (3.3.3)[66]. Les deux dernières sections proposent respectivement un retour réflexif sur la manière dont nous avons approché les acteurs ainsi que la prise des considérations éthiques liées à la thèse.

3.3.1 Observations présentielles

Pour chacun des deux projets, nous avons participé à deux rencontres publiques réunissant un bon nombre d'acteurs impliqués. Dans le cas de symfony, une première rencontre *Symfony Live*, au mois de juin 2009, se déroulait en français et réunissait environ 200 personnes. Une seconde rencontre également nommée *Symfony Live*, se déroulait également à Paris, en anglais et réunissait environ 400 personnes. Dans le cas de SPIP, nous avons participé à deux rencontres intitulées *SPIP-Party*, qui réunissaient chacune environ 50 personnes. La première, en juin 2009, s'est déroulée à Avignon tandis que la seconde s'est tenue à Gué du Loir, près de Vendôme[67]. Tel que décrit plus tôt ces *SPIP-Party* étaient beaucoup moins formels, et plus festifs, si on les compare aux rencontres de symfony. Au sein de SPIP, il existe également une tradition de «SPIP-Apéro », hebdomadaire. Bien qu'appelées apéros, ces rencontres sont plutôt l'occasion pour les acteurs d'échanger et de s'entraider sur leurs problèmes et leurs questionnements concernant la mise en place de SPIP. En particulier durant notre deuxième séjour à Paris (de janvier à mai 2010), nous avons participé à plusieurs de ces apéros qui ne réunissaient parfois cependant que trois ou quatre personnes.

Les notes d'observation de ces événements n'ont généralement pas constitué un matériau formel que nous citons dans notre thèse. Nous avons pour cela priorisé les entrevues ou

---

66 D'une certaine manière, notre approche ressemble au cadre méthodologique développé par Sack *et al.* (2006) pour l'analyse sociocognitive de la collaboration dans les projets de logiciels libres et open source. Pour ces auteurs, le développement distribué des logiciels libres est trop « fibreux » pour pouvoir être analysé à partir d'une seule perspective méthodologique. Les auteurs proposent par conséquent de combiner diverses méthodes telles que l'ethnographie, l'analyse textuelle (*text mining*), l'analyse des réseaux sociotechniques ainsi que différentes formes de visualisation pour comprendre les dynamiques collaboratives au sein de ces projets.

67 À titre anecdotique (et humoristique), notons que le moment le plus traditionnellement ethnographique de l'enquête a sans doute été notre participation au SPIP-Party de mai 2010. Cet événement se déroulait dans un troglodyte à Gué du Loir (France). Nous avions le choix de dormir dans une des grottes, ou bien d'installer notre propre tente, ce que nous avons choisi de faire. Durant cet événement, la journée se passait donc à prendre des notes sur les différents interventions et échanges entre les acteurs. Le soir venu, nous retournions dans notre tente et mettions en forme les observations faites durant la journée à propos de ces indigènes se réunissant dans une grotte !

l'analyse des traces en ligne. Ces observations ont par contre été cruciales pour mieux comprendre chacun des projets dans leur globalité et en particulier, pour nous intégrer dans la communauté, recruter des participants aux entrevues et identifier des éléments qui pourraient être intéressants à analyser plus en profondeur.

3.3.2 Entrevues semi-dirigées

Une dizaine d'entrevues semi-dirigées ont été réalisées pour chacun des projets étudiés. Ces entrevues se sont déroulées la plupart du temps en face à face mais quelque fois également par Skype, ou par la téléphonie IP, de façon à pouvoir enregistrer la conversation.

Une première série d'entrevues a été réalisée de juin à août 2009 et avait initialement pour objectif d'obtenir une première perspective générale des projets étudiés. Nous demandions par exemple au participant, à la participante de décrire le projet et son histoire. Dès les premières entrevues, cependant, des enjeux spécifiques liés au code source ont été discutés. Nous avons ensuite réalisé une seconde série d'entrevues entre février et mai 2010. Avant chacune des entrevues, une grille d'entretien personnalisée était préparée à partir de certains renseignements que nous avions collectés sur le participant, la participante, sur Internet, et en particulier, sur les interventions qu'il ou elle avait faites sur les listes de discussion et d'autres espaces en ligne. La grille d'entretien était donc personnalisée pour chacun-e des participants-es, et évoluait au fur et à mesure de l'analyse. Nous présentons en appendice B un exemple d'une telle grille d'entrevue. Les entretiens étaient généralement structurés de la façon suivante :

1) Présentation de la recherche;

2) Questions « contextuelles » concernant la formation du participant, de la participante, de même que son rôle, ou sa contribution, au sein du projet (SPIP ou symfony);

3) Questions visant à mieux comprendre le projet, et en particulier, les débats/controverses/ruptures marquants au sein du projet, et dont a été témoin le participant, la participante;

4) Questions sur la définition du code et du code source. Qu'est-ce que le code source, d'un point de vue concret et abstrait ?;

5) Différentes questions sur les appréciations (propreté, lisibilité, beauté) et les règles concernant l'écriture du code;

6) Dans certains entretiens, et en particulier dans les derniers, une plus grande insistance a été donnée à la description du processus d'écriture du code.

Dans la plupart des cas, nous avons tenté de relier les questions à des propos tenus par le participant, la participante, soit dans le passé (lors de rencontres physiques ou sur les forums en ligne), soit plus tôt dans l'entrevue. Ainsi, dans la plupart des entrevues, les participants utilisaient l'expression « écrire du code », alors que nous évitions nous-mêmes d'en faire mention. Nous relancions ensuite l'entrevue en posant une question du type « Tu parles d'écrire du code; comment t'y prends-tu pour écrire du code ? ». D'ailleurs, l'importance grandissante donnée à la catégorie de « l'écriture » dans cette thèse est en bonne partie due à ces références répétées à l'action d' « écrire du code ».

Cette façon de lier nos questions aux propos des acteurs a également été importante dans notre exploration des « qualités » du code (voir chapitre 6, section 2). Ainsi, très rapidement dans les entrevues, la plupart des participants utilisaient des qualificatifs tels que « propre », « lisible », « sale » ou « spaghetti » pour décrire certains morceaux de code (source). Plusieurs de nos questions d'entrevue subséquentes ont consisté à explorer ce que la personne interviewée entendait par ces adjectifs.

D'une manière générale, nous avons donc tenté de nous appuyer le plus possible sur les catégories des acteurs. Notons toutefois que deux catégories font exception à cette approche. D'abord, la catégorie du « code source » que les acteurs utilisent en fait assez peu dans le langage courant, au profit du terme « code », qui est généralement utilisé comme un diminutif du terme « code source ». Cet aspect sera amplement exploré dans le chapitre suivant. Une autre catégorie que nous avons souvent introduite dans les questions (sans qu'elle survienne d'elle-même) est celle de la « beauté ». Comme expliquée au chapitre 6, l'idée d'une « beauté du code » nous a beaucoup obsédés au début du doctorat, mais nous nous sommes rapidement aperçu que ce terme n'était pas vraiment utilisé par les acteurs pour qualifier le code. Nous avons cependant décidé d'interroger tout de même les acteurs sur cette idée d'une « beauté du code », ce qui nous a permis de constater une certaine distance des acteurs par rapport à cette idée, en même temps que de mettre en relation la catégorie de la beauté avec les autres qualificatifs mentionnés par les acteurs (« propre », « lisible », « sale », etc.).

Recrutement

La sélection des participants et participantes aux entrevues a été faite de façon itérative, à partir de nos observations et des entretiens précédents. Le principal critère qui a guidé ce recrutement était la diversité des rôles et des formes de contribution dans les projets étudiés. Nous avons en particulier cherché à rencontrer au moins une femme pour chacun des projets étudiés. Ce critère de recrutement s'appuie d'une part sur la problématique de la faible présence des femmes constatée dans les communautés de logiciel libre (Ghosh, Robles, et Glott, 2002; Lin, 2006a; Haralanova, 2010). D'autre part, ce choix se justifie d'une manière générale par la lecture des travaux féministes, et en particulier ceux de Star (1999), qui insiste sur la nécessité de considérer également les acteurs moins visibles de l'innovation technique, dont les femmes. C'est d'autant plus vrai en ce qui concerne l'interaction avec le code qui, comme Haralanova (2010) le propose dans son mémoire de maîtrise, est souvent le fait des hommes.

Également dans cette perspective féministe, il est intéressant ici de noter que certains acteurs, voire certains collègues, souhaitaient nous « aider » dans notre recherche, en nous dirigeant presque systématiquement vers les « codeurs » (catégorie que nous explorerons au prochain chapitre) ou les leaders de la communauté. Dans d'autres cas, certaines personnes avec qui nous avons réalisé des entrevues nous ont répété à maintes reprises que, bien qu'étant heureux (ses) de pouvoir nous aider dans cette recherche, ils ou elles n'étaient pas la meilleure personne pour parler du code, puisqu'ils ou elles n'y connaissaient rien[68]. Ce « malaise du code », pour reprendre l'expression de Simondon (1958) à propos de la technique, mériterait qu'on s'y attarde dans des études subséquentes (mais avec quelle méthodologie ?). Il est en effet pour le moins étonnant de constater que des personnes – souvent des femmes –, qui sont pourtant capables de participer à une entrevue sociologique d'une heure sur le « code source », affirment n'avoir rien à dire à ce sujet.

3.3.3 Analyse des traces de « négociations » en ligne

Outre ces entrevues semi-dirigées, notre principal matériau regroupe les multiples traces en ligne. Les espaces concernés par ce type d'analyse renvoient aux listes de discussions

68 C'était surtout le cas pour les acteurs et actrices de SPIP que pour ceux et celles de symfony.

électroniques, mais également aux autres dispositifs de communication électronique utilisés par ces projets : sites web, listes et forums de discussions électroniques, gestionnaires de bogues (« bug trackers »)[69], commentaires inclus dans le gestionnaire de versions du code source; et finalement, dans le code source lui-même. À certains égards, cette partie de notre démarche pourrait rejoindre l'approche d'une « ethnographie des traces » décrite par Geiger et Ribes (2010) dans leur analyse de la coordination entre des humains et des robots (*bots*) pour réagir aux actes de vandalisme sur Wikipédia :

> At its core, trace ethnography is a way of generating rich accounts of interaction by combining a fine grained analysis of the various « traces » that are automatically recorded by the project's software alongside an ethnographically-derived understanding of the tools, techniques, practices, and procedures that generate such traces (Geiger et Ribes, 2010, p. 119).

Geiger et Ribes soutiennent que la méthodologie d'une « ethnographie de trace » est similaire à l'esprit de Lucy Suchman, dans son analyse des interactions humain-machine. Ils notent toutefois que l'utilisation de l'observation vidéo peut difficilement être applicable pour saisir l'action des utilisateurs et de la technologie dans des phénomènes de réseau, tels que la lutte contre le vandalisme sur Wikipédia. L'ethnographie de trace consiste donc en quelque sorte à reconstituer l'action des utilisateurs et des dispositifs, à partir de l'analyse des traces en ligne.

Cependant, contrairement à l'étude Geiger et Ribes dont l'analyse est circonscrite à un corpus bien délimité de traces, il nous est pour notre part assez difficile de rendre compte d'une manière systématique de notre cheminement, étant donné l'éparpillement des espaces et des lieux où l'on retrouve des traces en ligne. Par exemple, pour le cas de SPIP, le site listes.rezo.net recensait 51 listes liées directement à SPIP (en date du 4 octobre 2010), et ce en plus de multiples blogues, un canal IRC que nous n'avons pas du tout analysé, ainsi que différents espaces de gestion des versions du code source ou de suivi des bugs (sur lesquels nous nous sommes particulièrement attardés dans notre étude). Cet éparpillement des lieux (virtuels) de conversations et d'interactions entre les acteurs de SPIP est d'ailleurs décrit par

---

69 Sur la question des « bug trackers », mentionnons ici un article de Joseph Reagle qui propose de considérer les « bug trackers » comme un espace public (Reagle, 2007). Voir également Shukka et Redmiles (1996) pour une analyse des formes d'apprentissage dans les logiciels de type « bug trackers ».

certains acteurs de SPIP comme la « galaxie SPIP[70] » (nous pourrions cependant faire une analyse similaire pour symfony). Compte tenu de cet éparpillement, il va donc de soi que notre analyse est nécessairement partielle et que nous ne prétendons pas ici donner un portrait exhaustif de l'ensemble des interactions ayant lieu dans les projets donnés. Cet éparpillement est d'ailleurs la partie de l'enquête qui rejoint le plus la problématique « multisite » à laquelle nous avons fait référence plus tôt.

Les « traces en ligne » dans notre cas ont été saisies de deux façons. D'abord, par un regard que nous pourrions qualifier de « flottant », dans ce sens que nous cherchions à comprendre le projet dans son ensemble et à repérer des cas particuliers à analyser. Ensuite, à partir de ce premier regard flottant, et nous inspirant des études STS, en particulier de la théorie de l'acteur-réseau, nous nous sommes attardés à cerner, puis à analyser plus précisément différentes « controverses », car celles-ci constituent un site privilégié où les acteurs déploient leurs propres catégories. Dans le cadre de notre analyse, nous donnons toutefois un sens large à la notion de controverse pour plutôt inclure ce que Goldenberg (2010) décrit comme des « négociations », c'est-à-dire des conversations argumentatives impliquant seulement quelques échanges de courriels[71]. À la différence de Goldenberg, toutefois, nous ne nous intéressons pas à analyser ces négociations pour elles-mêmes, mais nous nous en servons plutôt pour explorer la manière dont les acteurs déploient les catégories pertinentes à notre étude. En outre, les différentes « négociations » que nous avons analysées ont en commun de se rapporter à des évolutions et à des transformations concrètes du code source informatique ou à la négociation de règles et de conventions sur lesquelles les acteurs s'appuieraient ensuite dans leur fabrication collective du code source. Notons que dans ces dernières analyses, nous avons fait ressortir le caractère multisite en montrant que ces négociations s'appuyaient sur plusieurs dispositifs de communication.

---

70 Voir par exemple la page « Les sites de la galaxie SPIP » <http://tice.aix-mrs.iufm.fr/spip/Les-sites-de-la-galaxie-SPIP> et le site <boussole.spip.org> intitulé « Perdu dans la galaxie SPIP? » (sites consultés le 9 avril 2012).

71 Nous utilisons dans cette thèse le terme de « conversation » pour désigner empiriquement un ensemble de courriels qui se répondent l'un et l'autre. Le terme anglophone « thread » pourrait également être utilisé pour désigner cette unité analytique. Nous utilisons toutefois le terme « conversation » car c'est celui-ci qui est utilisé à la fois par les logiciels Gmail et Thunderbird. Le calcul du nombre de « conversations » ici présenté a été réalisé à l'aide du logiciel Gmail.

Entre ces deux types d'analyse, notre attention s'est également portée sur l'analyse de quatre listes de discussions – deux pour chacun des projets – sur une période d'un an, comprise entre 1er juillet 2009 et le 30 juin 2010. Pour chacun des projets, nous avons en effet choisi une liste dédiée au « développement » du projet (les listes symfony-dev et spip-dev) et à l' « usage » du projet (les listes spip-zone et symfony-user). Soulignons toutefois que le terme d'« usage » dans ce cas-ci est problématique, puisque la plupart des discussions sur ces dernières listes d' « usagers » concernent la manière d'étendre, ou de reconfigurer, les logiciels afin de les adapter à des usagers particuliers. Cette distinction problématique entre « développement » et « usage » sera approfondie dans les prochains chapitres.

3.3.4 Approcher les acteurs

L'approche des acteurs s'est faite graduellement et parfois difficilement au début. D'une manière générale, notre relation avec le terrain est cependant restée assez distante et nous n'avons pas établi des liens de proximité avec les acteurs concernés. Dans ce sens, notre relation avec le terrain rejoint la description que fait Licoppe de la position de l'analyste dans les études STS comme étant « nettement séparée de celle des agents impliqués dans les systèmes collectifs qu'il étudie » (Licoppe, 2008).

Dans le cas de symfony, nous avons d'abord contacté par courriel le chef du projet symfony, quelques jours avant la première rencontre publique à laquelle nous avons assisté. Nous n'avons dans un premier temps reçu aucune réponse à notre courriel. Lors de la conférence en question, nous l'avons ensuite approché personnellement pour lui expliquer très brièvement notre recherche et solliciter une entrevue. Il a répondu qu'il n'avait aucun problème avec le fait qu'une recherche prenne comme objet sa communauté, mais qu'à titre personnel, il n'avait pas beaucoup de temps à consacrer à ce projet. Quelques semaines plus tard, nous l'avons rencontré une première fois pour une entrevue, qui n'a duré que 45 minutes. Bien que les autres entrevues dans symfony étaient plus longues, elles étaient en moyenne plus courtes que celles de SPIP.

Dans le cas de SPIP, la communauté était d'emblée plus ouverte. D'abord, comme nous l'avons indiqué au chapitre 1, une étude ethnographique avait déjà été réalisée par Demazière, Zune et Horn (2007b; 2007a; 2006) et la communauté avait donc déjà une certaine habitude de cette démarche. Par ailleurs, les rencontres de SPIP étant beaucoup moins formelles et

réunissant également un nombre moindre de personnes, l'intégration à la communauté a été plus aisée. Très rapidement, nous avons pu connaître beaucoup mieux les différents acteurs de la communauté, en développant parfois une relation qui aurait pu relever de l'amitié, si ce n'avait été de la distance que nous souhaitions maintenir dans le cadre de la recherche. Ainsi, dans différentes occasions, nous avons eu l'opportunité de séjourner quelques jours chez certains des participants de SPIP. À d'autres moments, nous avons terminé une soirée au restaurant après un SPIP-Apéro, ou pris une bière en revenant d'un SPIP-Party. Ces rencontres ont donné lieu à des discussions plus en profondeur, qui nous ont permis de mieux comprendre les valeurs qui animaient cette communauté.

3.3.5 Considérations éthiques

Les espaces sur Internet librement accessibles, qui ont été pendant un certain temps considérés comme des espaces publics, sont de plus en plus l'objet de discussions éthiques. Ainsi, selon l'expression de Cardon (2008), on a de plus en plus affaire sur Internet à une visibilité en « clair-obscur » où des personnes se dévoilent parfois beaucoup, mais ont en même temps l'impression de ne le faire que devant des personnes en qui elles ont confiance, soit parce que l'espace où ils et elles interagissent n'est pas publiquement accessible, soit parce que les moteurs de recherche ne sont pas, ou plutôt n'étaient pas jusqu'à récemment, suffisamment perfectionnés pour permettre à autrui de retracer ces interventions. Cependant, avec des moteurs de recherche comme Google qui indexent de plus en plus profondément l'Internet, il devient facile de retrouver des informations sur des listes de discussions, ou des sites web, qui n'étaient pas initialement prévus à cette fin. Si une information, et en particulier une conversation, est accessible librement sur Internet, il n'est pas évident que la ou les personnes à l'origine de l'information acceptent que celles-ci puissent se retrouver dans un autre contexte. Bien qu'il ne semble pas y avoir de règles claires à respecter sur le plan éthique, une politique de base que nous avons respectée est d'éviter de fusionner des informations qui laisseraient apparaître la personnalité d'un individu.

Dans le cas des projets que nous avons étudiés, nous croyons cependant que ces difficultés éthiques sont moindres. D'une part, SPIP et symfony étant des projets de logiciels libres et à code source ouvert, nous pouvons supposer que l'ouverture des espaces de discussion a été depuis le début assumée en tant que telle, et que les acteurs qui participent à ces projets sont

pleinement conscients que leurs interventions sont sujettes à être lues par des personnes tierces. Ensuite, et peut-être surtout, de par sa nature, notre recherche ne s'attarde pas vraiment à des aspects intimes et personnels de la vie des individus. Finalement, notre recherche n'a pas non plus pour objectif de développer une critique des projets donnés, et encore moins des personnes qui sont impliquées dans ces projets, ce qui évite des risques de blesser ou de stigmatiser certains groupes ou individus. En bref, il nous semble que notre objet de recherche est relativement « neutre » d'un point de vue éthique.

Nous avons toutefois choisi de prendre certaines précautions. D'une part, nous avons choisi de conserver l'anonymat lorsque nous citons les entretiens, en utilisant plutôt une référence à l'entretien. Ainsi, **sf10** signifie l'entretien numéro 10 de symfony, tandis que **spip03** renvoie au troisième entretien de SPIP. Dans le cas des entretiens également, nous avons demandé aux participant-es de signer un formulaire de consentement si c'était possible. Dans le cas contraire (lors d'entretiens téléphoniques par exemple), nous leur demandions, au début de l'entrevue, d'indiquer explicitement avoir lu le formulaire de consentement et de signifier qu'ils consentaient à participer à cette entrevue.

Concernant les traces en ligne, nous avons tenté d'éviter de citer directement le nom d'une personne dans la thèse. Une autre avenue explorée a été de recourir à des captures d'images plutôt que de recopier le texte de l'information, ceci afin d'éviter qu'un moteur de recherche comme Google puisse retracer cette citation dans notre thèse.

Pour ce qui est finalement de la référence aux noms des projets étudiés (SPIP et symfony), les approches divergent dans le domaine, mais la publication du nom des projets semble davantage de mise dans les cas, comme pour cette étude, où les projets sont « en ligne », et de toute façon facilement repérables. De plus, les dirigeants et les acteurs-clés de chacun des projets ont été informés de notre étude et n'ont pas manifesté de réticences particulières quant à celle-ci. Pour ces raisons, mais également aux fins de clarté et pour donner crédit aux projets qui ont accepté d'être étudiés, nous avons décidé de conserver la référence au nom des projets.

3.4 Quelques notes sur l'analyse

3.4.1 De l'usage frustrant des logiciels d'analyse qualitative

Notre analyse a été marquée par un souci tourmenté et partagé entre un formalisme parfois excessif et le désir d'une interprétation plus libre. Ainsi, nous avons passé un temps considérable à tenter de procéder à une analyse formelle de nos données, à l'aide de logiciels d'analyse qualitative. Dans un premier temps, nous avons en effet recherché des logiciels « libres » qui pouvaient fonctionner sur Linux, tels que *WeftQDA* ou bien la suite logicielle *Cassandre*, développée notamment par Christophe Lejeune (2008). Ces logiciels se sont rapidement avérés décevants : WeftQDA est trop élémentaire pour nos besoins, tandis que Cassandre est encore à un stade de développement expérimental, ce qui impliquait une trop grande dépendance, au quotidien, face aux auteurs du logiciel. Pour un certain temps, nous nous sommes ensuite tournés vers *WiredMarker*, une extension du navigateur Firefox, qui permet de « marquer » certains passages de pages web, marquages qui peuvent ensuite être hiérarchisés pour former des thèmes plus larges, dans l'esprit de la théorisation ancrée. Un commentaire pouvait également être fait pour chacun des passages marqués, commentaire qui pouvait en quelque sorte tenir lieu de mémo. Cependant, *WiredMarker* n'a pas été conçu d'emblée comme un logiciel d'analyse qualitative et plusieurs fonctionnalités d'analyse qualitative sont absentes, notamment en ce qui concerne les outils de recherche tenant compte du codage.

Finalement, après un certain temps, alors que nous nous sentions « opprimés » par *WiredMarker*, nous avons décidé de nous tourner vers *MaxQDA*, un logiciel d'analyse qualitative propriétaire, mais dont le tarif étudiant était relativement accessible. L'usage de MaxQDA a donné des résultats intéressants jusqu'au moment de faire une mise à jour du logiciel, de la version 2007 à la version 2010, mise à jour qui s'est avérée catastrophique. Bien que cette version contenait des fonctionnalités intéressantes, celle-ci étant franchement « boguée ». Après quelques semaines d'utilisation, notre document s'est avéré corrompu : il devenait impossible de coder de nouveaux passages et certaines parties de l'interface disparaissaient !

En plus de l'utilisation des logiciels d'analyse qualitative, nous avons tenté de procéder à une analyse plus systématique des listes de discussions. Notre intention était surtout de procéder

par une analyse de type « lexicale », c'est-à-dire par recherche de mots clés. Beaucoup de temps et d'énergie ont été dédiés à tenter d'exporter les milliers de courriels de certaines listes de discussions, en fabriquant divers scripts PHP. Cette exercice a été intéressant car il nous a permis de nous remettre dans le bain de la programmation PHP et d'avoir une idée plus « sentie » de ce que constitue le code source en PHP. Par contre, après de longues tentatives dans notre projet d'importer les courriels dans MaxQDA, nous avons finalement décidé d'abandonner cette avenue.

3.4.2 Explorer et analyser le corpus : méthodes retenues

Ces frustrations par rapport aux logiciels d'analyse qualitative se sont progressivement dissipées lorsque nous avons décidé d'abandonner l'idée que les logiciels allaient faire l'analyse à notre place, pour plutôt ancrer notre démarche dans le processus d'écriture. Dans un certain sens, cette perspective rejoint les propos de Latour qui considère le texte comme l'équivalent fonctionnel du laboratoire : « C'est là où on fait des tests, des expériences et des simulations » (Latour, 2006, p. 217). Si l'usage des logiciels d'analyse s'est poursuivi, en revanche, l'analyse était beaucoup plus ancrée dans la rédaction et articulée dans un aller-retour permanent entre rédaction et exploration des données. Ainsi, nous pourrions dire que nous nous sommes servis davantage des outils d'« analyse » informatisés pour l'exploration du corpus, tandis que l'analyse proprement dite s'est plutôt faite dans la rédaction. L'exploration du corpus s'est surtout faite de la façon suivante :

– Navigation « flottante » des différents sites web liés aux projets symfony et SPIP, ainsi que de certaines parties du code source des projets donnés. Les recherches par mots-clés ont été principalement utilisées dans cette partie.

– Différentes recherches de type lexicographique, ou par mots-clés, des courriels envoyés sur quatre listes de discussions (près de 23 000 courriels au total. Voir le tableau 4.1). Cette recherche s'est faite à l'aide du logiciel de gestion de courriels Thunderbird, du service Gmail et de AntConc[72], un logiciel d'analyse de concordance. Certaines des discussions ont par la suite été ciblées pour faire l'objet d'une analyse plus approfondie.

72 <http://www.antlab.sci.waseda.ac.jp/software.html> (consulté le 6 décembre 2011).

– Ces discussions, de même que les notes d'observation et les transcriptions d'entrevues, ont été regroupées dans un fichier de type MaxQDA. Le marquage de certains passages, mais également la recherche par mots-clés ont également été utilisés de façon extensive pour analyser ce corpus.

La cohérence de notre analyse « multisite » pourrait être définie, d'une part, par la circonscription de notre analyse à deux terrains : SPIP et symfony. D'autre part, ce qui relie ces différents sites est surtout l'orientation de notre regard vers le code source, orientation qui s'exprimait notamment par le choix des mots-clés utilisés, tels que « code », « code source », « écriture », « commit », « conventions », « beauté », etc. Enfin, nous avons décidé de nous appuyer particulièrement sur les entrevues dans notre analyse.

3.4.3 La recherche qualitative, un processus itératif et rétroactif

Bien que le format de la thèse présente un développement linéaire (problématique, théorie, méthodologie, analyse), notre démarche a plutôt été ancrée dans un aller-retour permanent entre terrain, théorie et problématique. Cet aller-retour permanent s'est notamment exprimé dans le processus d'écriture, qui pourrait se décrire comme un cycle d'écriture et de réécriture des différents chapitres. Ainsi, nous n'avons pas rédigé dans un ordre linéaire les chapitres 1,2,3,4,5 et 6, mais plutôt procédé par cycles de d'itérations : 1,2,3 – 4,5,6 – 4,5 – 6 – 1,2,3 – 5,6. Au niveau conceptuel, notre démarche de recherche a ainsi procédé par de nombreux détours, en explorant des pistes qui ont ensuite été abandonnées[73]

Différents auteurs ont noté que cette démarche itérative ou rétroactive serait le propre de l'analyse qualitative. Millerand décrit par exemple dans sa thèse que la construction de son objet de recherche a nécessité des « allers et retours constants entre les observations de terrain, la consultation de la littérature et la formulation théorique » (Millerand, 2003, p. 109). Elle note d'ailleurs que la problématique de sa recherche a été rédigée dans sa forme finale à la toute fin du processus d'écriture de la thèse. Cette approche rejoint les propos de Deslauriers et Kérisit (1997) qui soutiennent que la recherche qualitative n'est pas linéaire, mais qu'elle est au contraire un aller-retour constant entre le terrain, la théorie et la

73 Par exemple, comme mentionné en avant-propos, et en introduction, nous avons été pour un certain temps préoccupé par la question de la « beauté » du code, qui a progressivement pris moins d'importance, au profit d'une attention plus précise portée à la notion de « code source ».

spécification de l'objet de recherche : « À mesure que progresse le travail simultané de collecte d'informations et d'analyse, l'objet de recherche se précise et les questions deviennent plus sélectives » (Deslauriers et Kérisit, 1997, p. 96).

Deslauriers et Kérisit (1997) notent toutefois un certain décalage entre le processus « conventionnel » d'écriture d'un rapport de recherche et la démarche effective de recherche. Les auteurs font ainsi remarquer que le modèle conventionnel d'écriture d'un rapport de recherche (ou d'une thèse) subvertit la chronologie du processus de recherche au profit d'une logique de formulation où l'objectif serait d'exposer la problématique étudiée (p. 103). Dans le modèle conventionnel d'écriture en sciences sociales, « Le souci de clarté et d'objectivité domine alors l'énonciation, loin du foisonnement du réel et des contradictions perçues ou réelles du terrain » (p. 103). Les auteurs notent toutefois que cette pratique d'écriture « conventionnelle » a cependant été remise en question en recherche qualitative, en particulier par l'anthropologie et le féminisme, sous le prétexte que cette forme de rapport de recherche (ou de thèse) ne rendrait pas compte du véritable déroulement de la recherche, et qu'elle occulterait le travail de création et de reconstruction de la réalité. Ces formes d'écriture se rapprocheraient davantage de l'essai humaniste que de l'article scientifique :

> La présence de l'auteur y est directement exprimée, à l'aide du « je » ou de l'anecdote. Ce rapport met en valeur un style propre à l'auteur, qui perçoit son travail de rédaction comme un travail de création littéraire (Becker, 1986, p. 105) [...] Il n'est plus seulement question de faire voir le phénomène social comme si on y était, mais de décrire le rapport du chercheur et de l'acteur social étudié, soit par une présentation du dialogue qui s'établit entre eux, soit par une écriture qui s'approche du roman ou de la poésie (Deslauriers et Kérisit, 1997, p. 104).

Deslauriers et Kérisit notent toutefois que cette forme d'écriture qui privilégie le retour sur soi a été souvent critiquée en sciences sociales. Les auteurs citent par exemple Bourdieu qui décrit cette démarche comme une « mode chez certains anthropologue américains [...] qui, ayant apparemment épuisé les charmes du travail sur « le terrain » se sont mis à parler d'eux-même plutôt que de leur objet d'étude » (Bourdieu et Wacquant, 1992, p. 52; cité par Deslauriers et Kérisit, 1997, p. 105). Pour Bourdieu, cette démarche d'écriture, sous un apparat faussement radical, ouvrirait la porte à un relativisme nihiliste et se situerait « à l'exact opposé d'une science sociale véritablement réflexive » (Bourdieu et Wacquant, 1992, p. 52; cité par Deslauriers et Kérisit, 1997, p. 105).

D'une certaine manière, le choix d'une forme d'écriture de thèse relève donc d'un certain positionnement épistémologique : plus scientifique d'un côté, par une présentation conventionnelle des chapitres de thèse, et par l'emploi du « nous »; plus humaniste d'un autre côté, en prenant une forme qui pourrait rejoindre celle du roman. Selon Deslauriers et Kérisit (1997), la majorité des recherches en sciences sociales se font encore aujourd'hui sur le mode conventionnel.

Pour notre part, nous avons décidé de recourir au style conventionnel, bien qu'ayant certaines affinités avec les approches féministes et anthropologiques préconisant le style inverse. Ce choix s'explique d'une part par son caractère conventionnel. Au moment de décider du choix de la forme d'écriture de notre thèse (en particulier de l'emploi du « nous » ou du « je »), nous avons consulté les thèses réalisées par nos collègues dans le passé (Millerand, 2003; Latzko-Toth, 2010; Goldenberg, 2010), qui se conformaient tous et toutes assez étroitement au style conventionnel d'écriture. Le choix de ce style conventionnel nous permettait également de nous concentrer sur l'exploration de notre objet de recherche, le code source, plutôt que l'exploration du style lui-même. D'une certaine manière, notre choix exprime également le désir d'ancrer notre thèse dans le pôle « scientifique » des sciences sociales. Dans ce sens, si nous partageons certaines préoccupations, voire certains concepts, avec les partisans d'une approche « humaniste » de l'écriture d'un rapport de recherche, il reste néanmoins que nous avons ici la prétention de « faire de la science » et de réaliser un travail d'ordre scientifique.

3.4.4 L'influence d'un parcours personnel

Il nous semble cependant important de nous situer dans cette étude. Cette section s'écrira à la première personne (au « je ») puisqu'elle aborde des aspects plus personnels. Avant d'entreprendre un « tournant » vers la recherche en communications, j'ai réalisé un diplôme de premier cycle en informatique et ai également travaillé pendant un certain temps dans l'industrie de l'informatique. J'avais donc déjà une bonne idée de ce en quoi consistait le « code source » ainsi que certains des aspects mis de l'avant dans cette étude. En outre, avant d'entreprendre des études en communications, j'ai moi-même réalisé quelques sites web à l'aide du langage de programmation PHP[74]. Si ce travail « pratique » n'est pas analysé

74 De plus, durant ma thèse, j'ai expérimenté avec PHP à plusieurs reprises, pour réaliser ma propre analyse, mais également pour aider ma compagne dans sa recherche.

formellement dans les chapitres suivants, il a en revanche participé à une meilleure compréhension générale du code source et des projets étudiés (écrits en PHP). Cette recherche opère donc en quelque sorte un « dialogue à trois » entre la théorie, les cas étudiés, mais avec une sensibilité particulière liée à ma formation et mon expérience passées en informatique.

Cette connaissance a priori de l'objet d'étude que m'a permis ma formation en informatique a des avantages et des désavantages. Parmi les avantages, il est évident que mon expérience et ma formation me permettent une compréhension plus aisée du vocabulaire et des concepts mis de l'avant par les acteurs. Cette situation a cependant un désavantage important, d'un point de vue sociologique, dans ce sens que j'ai parfois eu beaucoup de difficultés à me défaire de mes préjugés et de mes conceptions a priori. Le plus important dans ce cas concerne la définition du code source. Au début de l'enquête, j'avais en effet une idée assez précise de ce qu'était le code source. J'utilisais sans distinction, à la manière des acteurs, les termes code et code source. Ce n'est que plus tard, au fil de mes entrevues, que j'ai commencé à prendre conscience du caractère problématique de la notion de code source, ce qui a conduit à la réalisation du chapitre 4, consacré à cette question. Cependant, même après avoir réalisé une première version de ce chapitre, qui problématisait la notion de code source, on m'a fait remarqué certaines ambiguïtés entre la manière dont moi-même je définissais le code source et les définitions des acteurs[75].

De manière plus générale, l'un des principaux défis dans cette thèse a été de rester « collé » à la fois aux propos des acteurs, mais également au code source. Cet enjeu représentait un double défi. D'abord, comme mentionné plus tôt, ma formation informatique m'a amené, jusqu'à la fin de la thèse, à prendre certains aspects du code source pour acquis, en négligeant les propos des acteurs. D'autre part, il s'agissait également d'éviter l'erreur sociologique constatée en problématique, consistant à analyser toutes les dynamiques sociologique *autour* du code source, tout en laissant de côté l'analyse de l'artefact lui-même. Cependant, comment analyser le code source – un objet certainement plus approprié pour les sciences informatiques – tout en restant dans une perspective sociologique ? Finalement, la réalisation

75 À cet effet, j'aimerais remercier encore une fois Geneviève Szczepanik pour ses relectures et ses précieuses remarques méthodologiques.

d'une étude sur le code source pose de grands défis en terme de vulgarisation. Il s'agissait ici de ne pas oublier de définir certaines notions du monde de l'informatique, qui m'étaient familières, mais qui pouvaient être inintelligibles pour le lecteur ou la lectrice. Encore une fois, la relecture du document par des collègues et amies a ici été précieuse.

Une autre dimension importante à noter concernant mon parcours personnel a trait à mon expérience militante. Depuis presqu'une quinzaine d'années, je suis engagé plus ou moins activement dans différentes associations et différents projets, surtout québécois, mais également internationaux, liés au logiciel libre ou marqués par un militantisme « de gauche ». Au Québec, je suis par exemple membre d'organismes tels qu'Alternatives, Communautique, Québec Solidaire (un parti politique de gauche, féministe et altermondialiste) et j'ai participé à la fondation de Koumbit, un organisme dont la mission est de réunir des informaticiens socialement engagés, en particulier sur les questions de logiciels libres. Ce parcours militant fait que j'éprouve « naturellement » plus de sympathie pour le caractère militant du projet SPIP que pour celui plus commercial du projet symfony. Bien que j'ai cherché à conserver une certaine neutralité dans la plupart de mes analyses, il reste qu'à certains endroits, et en particulier en conclusion, on pourra voir assez clairement mes couleurs.

Finalement, un dernier aspect de mon parcours personnel concerne ma relation avec les projets étudiés. De par mon parcours militant, je connais SPIP depuis une bonne dizaine d'années, alors que j'ignorais l'existence même de symfony quelques semaines seulement avant de déposer mon projet de thèse (en février 2009). De plus, mon appartenance à Koumbit trouble d'une certaine façon ma relation avec ces autres projets. Koumbit est en effet une organisation qui s'est construite en bonne partie autour du développement du logiciel *Drupal*, qui est perçu comme un compétiteur, à la fois de SPIP et de symfony. J'ai moi-même réalisé quelques sites web avec Drupal durant les dernières années. De façon générale, j'ai évité de faire part d'emblée de ce parcours professionnel et militant, en particulier de ma connaissance de Drupal, bien qu'en fin de parcours, j'ai été davantage porté à le faire (auprès des acteurs de SPIP)[76].

76 Le « dévoilement » de mon expérience avec Drupal ne s'est cependant pas avéré problématique. Outre quelques moqueries amicales, les acteurs rencontrés en entrevues semblaient comprendre que l'objectif de ma recherche était surtout lié à l'avancement des connaissances, et non à la promotion d'une technologie particulière.

D'une manière générale, mon parcours personnel et ma sensibilité informaticienne ont certainement influencé le choix de mon sujet de recherche, de même que ma capacité à dialoguer avec les acteurs à partir de leur propre vocabulaire. Cependant, il me semble important de souligner encore une fois qu'il s'agit ici d'une thèse en sciences sociales. D'une certaine manière, l'un des objectifs de cette thèse est de montrer que cet objet de recherche, le « code source », est digne d'être étudié dans une perspective des sciences sociales, et qu'il peut l'être par une chercheure, ou un chercheur, sans formation préalable en informatique.

3.4.5 À propos des notions de code et de code source. Une première clarification.

L'un des premiers étonnements dans notre enquête a été de constater que la définition de ce qui constituait le « code source », voire la pertinence même d'utiliser cette notion dans le cas des projets étudiés, était problématique pour certains participants à notre recherche. Tel qu'expliqué dans le chapitre précédent, nous avons reçu une formation universitaire en informatique et avions par conséquent une idée assez claire de ce qui constituait le code source. Cependant, nous nous sommes rapidement aperçu (avant même la première entrevue, dans le cas de SPIP) que cette notion était franchement problématique pour quelques participants. Ainsi, lors de la soirée ayant précédé notre première entrevue dans SPIP, nous discutions de façon informelle autour d'un verre, lorsqu'un participant nous a indiqué que la notion de « code source » ne faisait pas sens dans le monde de PHP, puisqu'il n'y avait pas de notion de *code objet.* PHP est en effet un langage interprété – aussi appelé un langage de script. Il s'agit donc d'un code qui est exécuté « directement » et qui ne nécessite pas une phase de compilation, traduisant le code source – écrit dans un langage humainement compréhensible – en un code objet, ou exécutable – spécifié dans le langage binaire de la machine et propre à être exécuté. Dans le cas d'un code écrit dans le langage PHP, notre interlocuteur préférait plutôt de parler de *code script* à la place de *code source*.

Cette épisode de remise en question radicale de notre objet d'étude nous a un peu troublé[77]. Nous pensions alors avoir choisi un mauvais terrain pour aborder la question du code source. Cependant, sans occulter cette problématique du code source, il nous semblait néanmoins

77 Il nous semble d'ailleurs que l'intention de notre interlocuteur à ce moment était précisément de nous provoquer et de nous remettre en question. Si c'est le cas, cette démarche a eu les résultats escomptés et nous tenons à l'en remercier !

exister, dans le cadre de SPIP, un objet qui correspondait à notre idée générale du code source que nous avions défini dans notre projet de thèse comme un « ensemble de documents textuels, qui contiennent des instructions spécifiant formellement le fonctionnement d'un logiciel ou d'un système informatique[78] ». De plus, comme nous le verrons dans la suite de notre analyse, plusieurs des acteurs de SPIP qui ont participé à nos entrevues n'éprouvent pas cette réticence conceptuelle face à la notion de code source, qui est même utilisée dans certains documents officiels du projet. Ainsi, on retrouve par exemple sur le site de SPIP une page décrivant l'historique du projet où il est écrit (nous soulignons) :

> De février 2002 à août 2005, le **code source** de SPIP était déposé dans et géré par un serveur CVS. [...] À partir du 21 août 2005, SPIP quitte son environnement de développement sous CVS pour passer sous SVN[79].

Clairement donc, malgré les affirmations troublantes de notre premier interlocuteur, il semble que la notion de « code source » renvoie à quelque chose de bien réel dans les projets étudiés. Cependant, cette première discussion informelle nous a amenés à prendre plus au sérieux la nécessité de bien définir le code source et de prendre en compte les définitions des acteurs. Plus important, ces discussions informelles nous ont permis d'explorer des définitions du code source très générales, qui renvoyaient à des aspects que nous n'avions nous-mêmes pas envisagés au départ. Nous avons ensuite décidé de suivre cette piste, de façon à prendre la définition même du code source comme objet d'étude, avant d'aborder le code source comme artefact concret. Ainsi, par la suite, nous avons insisté davantage sur la question de la définition du code source, approche qui s'est faite plus systématique dans les dernières entrevues.

Une autre manière de considérer la place du code source dans le vocabulaire des acteurs est d'analyser les listes de discussions. Le tableau 3.2 montre, pour chacune des quatre listes que nous avons retenues dans notre corpus, le nombre de conversations[80] ayant respectivement les

---

78 Projet de thèse soutenu en février 2009.

79 <http://core.spip.org/projects/spip/wiki> (consulté le 29 juin 2011).

80 Comme indiqué plus tôt dans le chapitre, le terme de « conversation » ne renvoie pas ici au courant de l'analyse de conversations. Il renvoie plus simplement au terme de « conversation » utilisé dans les logiciels Gmail et Thunderbird pour décrire une suite de courriels se répondant l'un à l'autre. Le calcul du nombre de « conversations » ici présenté a été réalisé à l'aide du logiciel Gmail.

mots « code », « code source » et la combinaison des mots « code » et « source ». Cette petite analyse lexicale fait clairement ressortir que le terme « code source » est effectivement utilisé dans les conversations entre les acteurs, bien que de façon assez occasionnelle. En revanche, le terme « code » est nettement plus utilisé que celui de « code source ».

Tableau 3.2 : Référence aux termes « code » et « code source » dans les listes de discussions de SPIP et de symfony[81]

		Nombre de « conversations »			
	Nombre de courriels	**Nombre de conversations**	**Avec les termes « code source » ou « source code » dans un même message**	**Avec « code » ET « source » dans un même message**	**Avec le terme code dans un message**
spip-dev	4 089	841	14	27	230
spip-zone	5 316	1 378	12	25	285
symfony-dev	1 675	365	8	22	124
symfony-users	11 861	3 355	36	88	805
Total	**22 941**	**5 939**	**70**	**162**	**1 444**

Une hypothèse simple permettant d'expliquer le faible usage du terme « code source » est que les acteurs utilisent simplement le terme « code » comme diminutif pour désigner ce qui est compris, au sens général, comme étant le « code source ». Quelques extraits de nos entrevues tendent à confirmer cette hypothèse (les parties en italique correspondent à nos interventions dans les entrevues) :

> *Est-ce que tu fais une distinction entre le « code » et le « code source »?*
> C'est une bonne question... Je t'avoue que je ne me l'étais jamais posée. Euh, le code... et le code source... Pour moi, c'est vraiment la même chose. J'ai jamais fait de distinction dans mon esprit sur ces deux formulations (sf08).

> *Fais-tu une différence entre « code » et « code source » ?*
> Pour moi, ça reste la même chose, puisque du point de vue de l'application, on a le code source. Le code qu'on écrit, ce sera le code source de l'application. [...] Pour moi, je ne fais pas de différences entre le code et le code source (sf07).

81 Courriels envoyés sur les listes entre le 1er juillet 2009 et le 30 juin 2010. Calcul réalisé avec Gmail.

Une fois bien établis les contours de l'objet que les acteurs définissent comme étant le code source, nous considérerons également comme étant le code source, ce que les acteurs désignent par le code, en particulier dans l'usage d'expressions telles que « le code qu'on écrit », « mon code » ou bien « le code de SPIP », « le code de symfony », etc. Pour l'instant, attardons-nous encore davantage aux différentes manières dont les participants à nos entrevues définissent plus formellement le code source. C'est ce que nous ferons dans le chapitre suivant.

CHAPITRE IV

LE CODE SOURCE DÉFINI PAR LES ACTEURS

> Le code, c'est à la fois ce qui fait que le programme tourne et ce qui fait que les gens se comprennent, ils écrivent quelque chose en commun (spip03).
>
> En fait, le code source je pense, c'est aussi celui sur lequel on va travailler, qui est malléable, et qu'on va faire évoluer, qu'on va pétrir pour faire évoluer le logiciel final (spip11).

L'objectif de ce chapitre est de répondre à notre première question spécifique de recherche : *Comment les acteurs définissent-ils le code source et en particulier, dans quelle mesure cet artefact peut-il être appréhendé comme un écrit ?* Comme noté en problématique, nous montrons dans ce chapitre que les frontières définitionnelles et « artefactuelles » de la notion de code source sont encore floues et que la délimitation de ces frontières est articulée à des enjeux politiques. En nous appuyant sur les entrevues réalisées avec les acteurs des projets étudiés, nous faisons tout d'abord ressortir différentes définitions « générales » que les acteurs donnent au code source, de même que certaines métaphores utilisées par les acteurs pour le décrire. Le chapitre se poursuit ensuite par la description concrète de l'artefact ou des artefacts désignés par les acteurs par le terme « code source ». Nous terminons en décrivant comment, pour quelques-uns des acteurs rencontrés, la définition même du code source est articulée à certains enjeux d'ordre politique. Ces descriptions et analyses permettent de saisir les contours de ce qui constitue l'artefact code source.

4.1 Formes et statuts du code source dans les projets étudiés

Cette première partie du chapitre présente les différentes formes et statuts que prend le code source dans les projets étudiés en analysant l'objet, ou les objets, désignés par les acteurs par les termes « code source de SPIP » ou « code source de symfony ». Il ne s'agit pas ici de présenter de façon extensive toutes les formes que peut prendre le code source, mais plus simplement de mettre de l'avant une certaine diversité de ces formes et statuts dans les projets

étudiés, ainsi que leur interrelation. En outre, l'accent est mis sur les aspects qui ressortent des entrevues et auxquels nous faisons référence dans les chapitres suivants.

Mentionnons d'emblée que cette analyse des formes et statuts que prend le code source ne relève pas que d'un pur formalisme. Le statut d'un artefact comme le code source et symétriquement, la définition même de cette notion, sont articulés à une dimension politique. Il s'agit entre autres de savoir si la création de tel ou tel artefact peut être assimilée au « codage » (*coding*), une activité hautement valorisée dans le monde du logiciel libre. En d'autres termes, la manière dont est appréhendée la définition du « code » et du « code source » peut être liée à la manière dont sont valorisées certaines activités au détriment d'autres. Cet aspect plus politique lié à la définition du code source sera cependant abordé plus précisément dans la dernière partie du chapitre.

4.1.1 « Le code source [...] c'est celui que je peux télécharger »

> Si je prends symfony, pour moi, le code source de symfony, c'est celui que je peux télécharger pour l'avoir chez moi et pour après l'utiliser (sf05).

Dans le cadre de symfony, certains participants à nos entrevues ont mentionné que le code source de symfony, est « celui que je peux télécharger » (sf05) ou encore, que « quand on télécharge symfony, on télécharge les sources de symfony » (sf07). Pour débuter notre analyse du code source, appuyons-nous sur ces propos. La figure suivante (4.1) montre un extrait de la page de téléchargement de symfony :

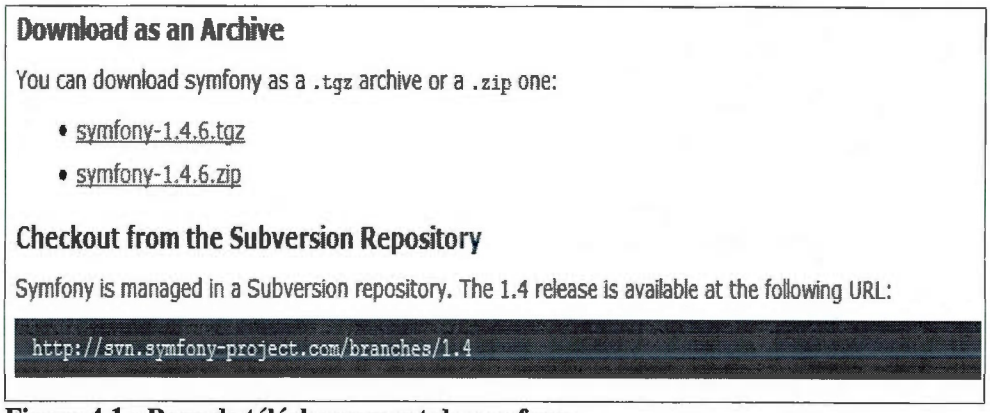


Figure 4.1 : Page de téléchargement de symfony
<http://www.symfony-project.org/installation/1_4> (consulté le 27 février 2012).

Cette figure est intéressante à analyser pour plusieurs raisons. D'abord, notons au passage que cette page est disponible uniquement en anglais, aspect que nous approfondirons au chapitre suivant. Ensuite, plus pertinent pour la présente analyse, on peut constater qu'il n'y a aucune mention des termes « code » ou encore « code source » dans cette figure. Il faut donc s'appuyer (en partie) sur les extraits d'entrevue cités plus tôt, et prendre pour acquis qu'il s'agit effectivement du code source de symfony. En effet, comme mentionné dans le chapitre précédent, le langage PHP – utilisé dans les projets comme symfony et SPIP – est un langage *interprété*, ce qui signifie que le code source est directement exécuté, au lieu de passer par une phase de compilation qui créerait un autre type de code, le code exécutable[82]. En d'autres termes, télécharger symfony est ici synonyme de télécharger le code source de symfony.

La figure 4.1 montre également trois manières de télécharger symfony, soit à partir d'une archive .zip, ou d'un fichier .tgz, deux technologies d'archivage distinctes, la première étant davantage utiliseé dans les systèmes Windows de Microsoft, et la seconde davantage utilisée pour les systèmes d'exploitation Unix. Une troisième manière d'obtenir symfony est de retirer le code source depuis le dépôt *Subversion* (« *Checkout from the Subversion repository* »), aspect qui sera abordé plus loin dans le chapitre.

Dans le cas de SPIP, nos entrevues ne permettent pas de faire ressortir une affirmation aussi spécifique indiquant que le code source serait « celui qu'on peut télécharger ». Cependant, un des participants affirme que « le code source c'est l'ensemble du logiciel SPIP » (spip07). La figure suivante (4.2) montre la page de téléchargement de SPIP. Comme dans symfony, SPIP peut être téléchargé depuis un fichier .zip. Il peut également être obtenu en récupérant un fichier permettant l'installation automatique. La « version de développement » est également disponible en téléchargement :

---

82 Dans le cadre de notre mémoire de maîtrise qui abordait plus spécifiquement les liens entre l'idéologie du logiciel libre et l'activité technique elle-même, nous avions mentionné qu'un des acteurs avec qui nous avions réalisé une entrevue préférait les langages interprétés parce qu'il se sentait plus « libre », dans le sens de logiciel libre (Couture, 2007).

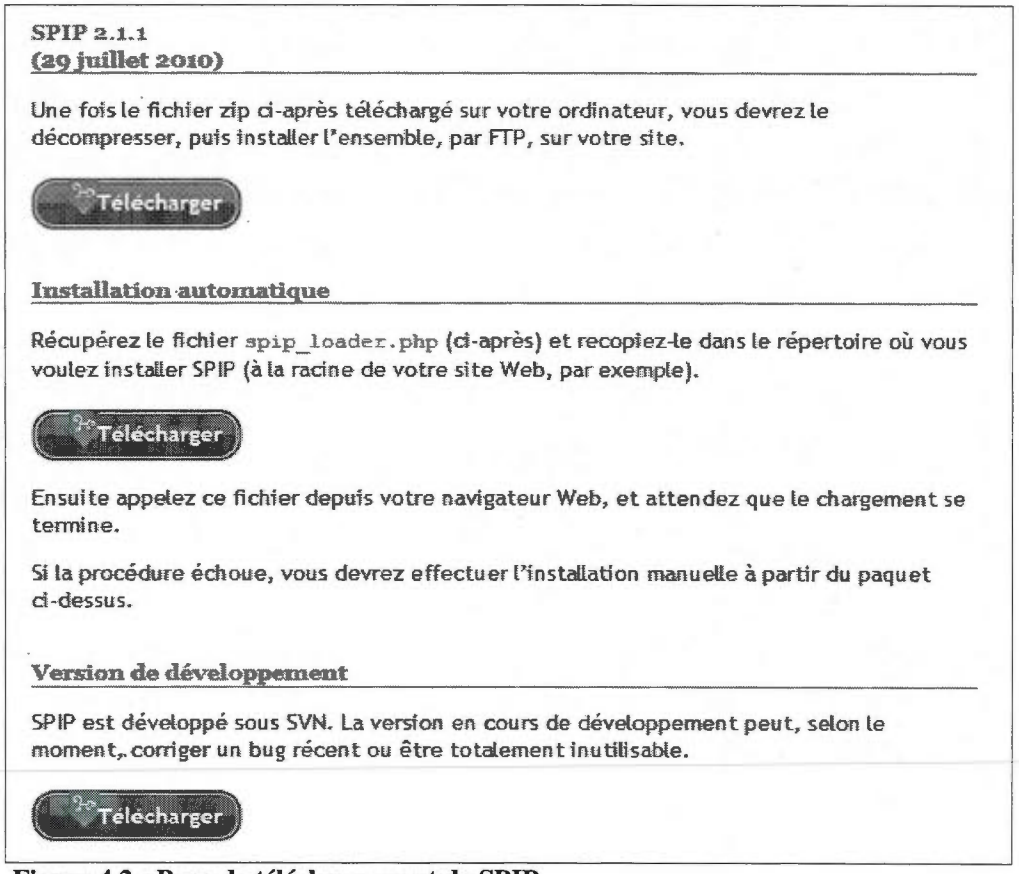


Figure 4.2 : Page de téléchargement de SPIP
<http://www.spip.net/fr_download> (consulté le 27 février 2012).

Nous avons fait l'exercice de télécharger et d'analyser les fichiers .zip de symfony et SPIP. Le tableau 4.1, à la page suivante, présente quelques chiffres qui décrivent le contenu de cette archive – c'est-à-dire le code source de chacun des projets.

Tableau 4.1 : Quelques chiffres pour décrire la complexité du code source[83]

Caractéristiques quantitatives	symfony 1.4.10		SPIP 2.1.8	
Taille	14,334 Mb		14,015 Mb	
Nombre de répertoires	788		99	
Nombre total de fichiers	2 924		1 305	
Nombre de fichiers par type (extensions)	.php	2 104	.php	714
	.dat	331	.gif	187
	.yml	157	.png	178
	.xml	102	.html	146
	.png	68	.js	30
	.[84]	30	.txt	17
	.mod	23	.css	15
	.sf	19	.jpg	7
	.txt	17	.xml	6
	.ini	8	.ttf	2
	autres	65	Autre	3
Nombre de lignes de code – fichiers .php	302 803 (12 080 pages[85])		223 161 (8 916 pages)	
Nombre de lignes de commentaires	110 773 (4 422 pages)		23 179 (925 pages)	

Ces chiffres montrent tout d'abord une certaine complexité dans l'organisation du code source qui nécessite, dans chacun des deux projets, des centaines de dossiers et de fichiers distincts. Ensuite, la comparaison entre les deux projets montre un nombre de répertoires presque huit fois plus élevé dans symfony que dans SPIP, et un nombre de fichiers deux fois plus grand dans symfony que dans SPIP et ce, pour une taille similaire. Ces différences renvoient à des styles différents d'organisation du code source dans les projets étudiés, styles que nous

83 La taille, le nombre total de fichiers, ainsi que le nombre de fichiers par type ont été obtenus avec le logiciel *Simple Directory Analyser*. Le nombre de répertoires a été obtenu avec l'explorateur *Windows*. Le nombre de lignes de code ainsi que le nombre de lignes de commentaires ont été obtenus avec le logiciel *LocMetrics*. L'analyse a été réalisée sur le code source des versions 1.4.10 de symfony et 2.1.7 de SPIP.

84 Aucune extension.

85 Ce chiffre est donné à titre d'illustration. Une page équivaut ici à 25 lignes de texte, soit l'équivalent d'une page de notre thèse de doctorat.

aborderons au chapitre 5, en particulier dans l'analyse de certaines controverses autour de l'organisation du code source.

Dans les deux cas, la majorité des fichiers portent l'extension .php, ce qui signifie qu'il s'agit de fichiers textes écrits dans le langage PHP. La figure 4.3 montre un extrait de l'un de ces fichiers portant l'extension .php pour le cas de SPIP.

```
178   // Convertir dates a calendrier correct
179   // (exemple: 31 fevrier devient debut mars, 24h12 devient 00h12 du lendemain)
180
181   // http://doc.spip.org/@change_date_message
182   function change_date_message($id_message, $heures,$minutes,$mois, $jour, $annee, $h
183   {
184           $date = date("Y-m-d H:i:s", mktime($heures,$minutes,0,$mois, $jour, $annee)
185
186           $jour = journum($date);
187           $mois = mois($date);
188           $annee = annee($date);
189           $heures = heures($date);
190           $minutes = minutes($date);
191
192           // Verifier que la date de fin est bien posterieure au debut
193           $unix_debut = date("U", mktime($heures,$minutes,0,$mois, $jour, $annee));
194           $unix_fin = date("U", mktime($heures_fin,$minutes_fin,0,$mois_fin, $jour_fi
195           if ($unix_fin <= $unix_debut) {
196                   $jour_fin = $jour;
197                   $mois_fin = $mois;
```

Figure 4.3 : Contenu d'un fichier .php de SPIP
Fichier ecrire/action/editer_message.php (SPIP 2.1.8). Fichier visualisé avec le logiciel *Netbeans*.

Cette figure montre bien la forme typique que prend le code source dans les projets étudiés. Dans le cas de SPIP, ce « texte » est écrit dans utilisant des « mots » inspirés du français, tels que « change_date_message », ou bien $heures, $minutes, etc. Certains passages, précédés par des '//' sont rédigés en français, par exemple « Vérifier que la date est bien postérieure au debut » (ligne 192). Il s'agit ici de commentaires. Dans le cas de symfony, c'est l'anglais qui est utilisé.

Si les fichiers de type .php comptent pour la majorité des fichiers de chacune des archives, ils n'en forment cependant pas la totalité. D'autres fichiers portent des extensions .html, ou .txt ce qui signifie qu'ils sont écrits dans le langage HTML, ou alors, simplement en format texte. Également, on retrouve dans ces archives de fichiers différentes images (.gif, .png, .jpg), qui constituent par exemple des icônes pour les liens, boutons ou encore, le logo de chacun des projets. Là encore, on voit bien les difficultés à définir le code source uniquement par son

caractère écrit. Cette analyse nous amène déjà à considérer que le code source n'est ni limité à un langage particulier, ni à des fichiers textes.

L'analyse de ce seul code source – « celui que je peux télécharger » – est cependant insuffisante pour rendre compte de l'ensemble des artefacts considérés par les acteurs comme du code source. Dans les deux projets étudiés, ce code source disponible en téléchargement sur la page officielle du site est désigné comme le « coeur » du projet, ou encore, le « *core* ». Il s'agit en fait uniquement du code source qui a été décrété à un moment donné, et par certaines personnes – le *core team* ou encore l'équipe de cœur[86] – comme étant une version *stable*. Dans le chapitre 6, nous appellerons ce code source – disponible en téléchargement – le code source *autorisé,* en nous attardant aux différents droits et autorisations avec lesquels ce code source autorisé est articulé. Autour de ce code source « autorisé », on peut noter une multitude de lieux et d'espaces où sont partagés, voire discutés, des morceaux de code source sous différents formes et statuts.

4.1.2 Le code source des plugins

> Un plugin de symfony ne fait pas partie du code source de symfony, il fait partie du code source d'une application qui est développée par dessus en fait (sf07).

Les *plugins* constituent pour chacun des projets un endroit privilégié où l'on retrouve du code source. Les plugins sont des modules d'extensions qui permettent d'ajouter des fonctionnalités au noyau du logiciel, voire de reconfigurer certaines parties. Ces plugins sont souvent créés par des développeurs de la communauté, souvent externes à la « *core team* ». En date du 27 août 2010, on retrouvait ainsi 255 plugins dans le projet SPIP et 1 103 plugins dans le projet symfony.

---

86 Le « core team », aussi appelé « équipe de coeur », regroupe pour chacun des projets une dizaine de personnes qui forment ce que Cardon (2005) appellerait le « noyau des innovateurs ». Le rôle de cette équipe est précisément de gérer et développer cette partie du code source nommé le « core », en plus de donner une orientation générale au projet dans son ensemble. Nous verrons au chapitre 6 que le membership au « core team » est articulé à différents droits et autorisations en ce qui a trait à l'écriture et à la modification du code source du core. Mentionnons finalement que l'équipe de cœur, ou « core team », est également désignée dans le projet SPIP par l'appellation « SPIP Team » ou simplement « La Team ».

Bien que le développement de plugins soit particulièrement important aujourd'hui, le « système de plugins » n'a pas toujours existé dans les deux projets. Dans le cas de SPIP, le système de plugins est apparu assez tardivement dans l'évolution du logiciel, à la version 1.9 plus précisément, soit en 2005, environ quatre ans après la création du logiciel. Le système de plugins y était vu comme une manière de contrer la possibilité d'un « *fork* », c'est-à-dire d'une division de la communauté pour créer une nouvelle version du code. Il faut dire en effet que la communauté SPIP a été particulièrement troublée par un épisode datant des années 2004 - 2005, lors duquel certaines institutions de l'administration publique française avaient réalisé un « *fork* » de SPIP, en développant un nouveau logiciel, nommée SPIP-Agora qui était basé sur le code source de SPIP. Bien que l'expérience SPIP-Agora s'est finalement avérée un échec, la possibilité d'un nouveau *fork* reste encore une préoccupation bien présente pour les acteurs de SPIP. C'est notamment dans cette perspective que le système de plugins a été développé, de façon à permettre à des parties tierces de pouvoir modifier le fonctionnement du logiciel, sans nécessairement « partir avec tout le projet ». L'un des acteurs de SPIP compare ainsi les plugins avec des pièces d'un puzzle qui permettent d'être reliées les unes aux autres :

> Donc puisqu'on a réalisé un ensemble de pièces de puzzle, qui fonctionnent les unes, les autres mais qui sont codées de manière indépendante, tu peux dans tous les cas en extraire une, et la remplacer par une pièce à toi. Pas obligé de partir avec tout le projet, de *forker* tout le projet, pour en modifier un partie (spip01).

L'introduction des plugins a eu une influence importante dans le projet. Elle a permis de décentraliser davantage la fabrication du code source en encapsulant certaines fonctionnalités au sein des plugins, et en étant plus libérale en ce qui concerne l'octroi des autorisations de modification de ces parties du code source (voir chapitre 6 pour une analyse de la mise en œuvre des autorisations liées à la modification du code source). Cette dynamique évite en outre aux responsables du projet d'avoir à évaluer puis valider toutes les possibilités de modification pour plutôt se concentrer sur un ensemble précis de code source – le *core*. L'architecture des plugins permet également à un plus grand nombre de personnes de participer à l'écriture collaborative du code source :

> Ça permet justement toute une variété de propositions à apparaître, et comme les gens collaborent sur les propositions, il y en a un qui la lance, l'autre qui la complète, l'autre qui va la réécrire, etc., ça fait émerger des compétences d'écriture, à la fois d'écriture de code, d'écriture collaborative, d'utilisation de SVN. Les gens apprennent à se connaître, à écrire du code ensemble (spip01).

Le projet symfony possède également un système de plugins. Comme dans SPIP, le système de plugins n'a pas été mis en place dès la création du projet, mais elle est arrivée dans une phase assez précoce du développement du projet.

Dans chacun des projets étudiés, les plugins sont répertoriés sur un site web de type « web 2.0 », où l'on retrouve différentes informations concernant le plugin en question, dont une description, de même que la possibilité de commenter et évaluer le plugin.

Tout comme pour le code source du projet, il est possible d'obtenir le code source des différents plugins soit par l'intermédiaire du système de gestion des versions, soit en téléchargeant le plugin sous une archive .zip directement depuis sa page web. Lorsqu'on télécharge un plugin – dans SPIP et dans symfony – on télécharge également le code source du plugin puisque, nous l'avons déjà écrit, dans PHP, le code source est directement interprété. Par exemple, le plugin « Gestion d'associations » mentionné à la figure 4.4 constitue une archive .zip qui contient 14 répertoires, 177 fichiers dont 96 portent l'extension .php[87].

87 Plugin « Gestion d'associations », version 2.0 (<http://files.spip.org/spip-zone/Association_2_0.zip> - consulté le 27 février 2012). Le nombre de répertoires et de fichiers est obtenu à l'aide de l'explorateur Windows, tandis que le nombre de fichiers portant l'extension .php est obtenu à l'aide du logiciel *LogMetrics*.

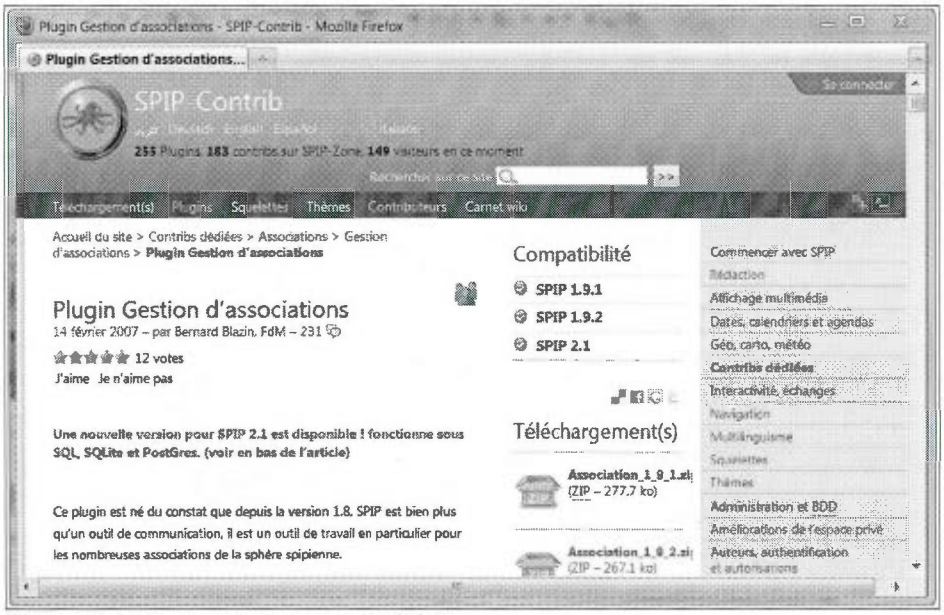


Figure 4.4 : Page d'un plugin de SPIP
<http://www.spip-contrib.net/Plugin-Gestion-d-associations> (consulté le 30 mars 2011).

Pour les acteurs, si ces fichiers composant les plugins sont considérés sous l'appellation « code source », ils ne possèdent pas le même statut que le code source du projet. Un acteur de symfony explique la manière dont ces statuts se distinguent :

> Alors, moi je distingue le code source de symfony, qui est développé par la *core team*, et puis les plugins, qui sont des sources qui sont développées par la communauté, par des développeurs PHP majoritairement externes à la *core team*, développeurs qui utilisent symfony dans leur entreprise. Pour moi, il y a cette distinction quand même. Un plugin de symfony ne fait pas partie du code source de symfony, il fait partie du code source d'une application qui est développée par dessus en fait (sf07).

Cette citation met de l'avant la manière dont l'appartenance ou non du code source au projet est articulée au fait d'être développé ou non par la *core team*. Elle nous introduit aussi à l'existence de certaines formes d'autorité et d'autorisation dans l'écriture du code source, sur lesquelles nous reviendrons au chapitre 6.

4.1.3 Squelettes de SPIP et fichiers de configuration YAML

Un aspect particulièrement intéressant de SPIP concerne ce qu'on appelle les squelettes ou encore, le « langage de squelettes ». Le langage de squelettes est un langage réalisé par la communauté SPIP elle-même, qui permet de mettre en page les différents éléments d'un site. Les squelettes utilisent ce que les gens désignent plus ou moins formellement le « langage SPIP ». Ce « langage de squelettes », qui est spécifique à la communauté SPIP, est une sorte d'extension du langage HTML, ce qui inclut quelques opérations supplémentaires, notamment ce que les acteurs appellent des « boucles », qui sont en quelque sorte des commandes à la base de données permettant d'afficher une série d'informations. Nous retrouvons à la figure 4.5 un exemple de ce « langage de squelettes » de SPIP.

```
1  #CACHE{3600*6}
2  #HTTP_HEADER{"Cache-Control: max-age=3600, must-revalidate"}
3  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"  "
4
5  <html xmlns="http://www.w3.org/1999/xhtml" dir="#LANG_DIR" lang="
6  <head>
7    <title>#NOM_SITE_SPIP - <:poster_message:></title>
8    <INCLURE{fond=inc/inc_head}{lang}>
9  </head>
10
11 <body>
12 <div id="page">
13   <INCLURE{fond=inc/inc_banner}{lang}>
14   <INCLURE{fond=inc/inc_navbar}{lang}{no_context=true}>
15
16   <!-- main -->
17   <div id="main">
18
19     <!-- main: content -->
20     <div id="content">
```

Figure 4.5 : Squelette de SPIP
<http://zone.spip.org/trac/spip-zone/browser/_squelettes_/a-brest/forum.html> (consulté le 27 février 2012). Comme son nom l'indique, ce squelette est utilisé pour le fonctionnement du site a-brest.net.

Le langage de squelettes est une particularité importante de SPIP qui explique pour certains la raison de son succès. C'est en particulier le fait que le langage de squelettes constitue un niveau intermédiaire entre le PHP et le HTML qui le rend « plus accessible que le PHP » (spip01) et qui permet une participation plus étendue à la reconfiguration de SPIP. Une actrice décrit ainsi le langage de squelettes, ou de boucles, de SPIP :

> SPIP, ce qui est extraordinaire, quand justement tu ne codes pas, tu vas réussir à faire des choses, comme si tu codais. Parce que ça a été intelligemment pensé par d'autres. Tout a été fait de manière à ce que même si tu ne comprends rien, tu vas pouvoir t'en sortir, quoi. Parce que tu as un langage simplifié de boucles qui permet de faire ce que tu fais avec du PHP, mais simplement de manière accessible, il y a un couche d'abstraction, en fait, qui est venue pour simplifier tout. Donc, du coup, ça devient accessible, et on n'est plus dans un domaine élitiste en fait, où il faut avoir trois ans de code dur pour comprendre, pour faire un truc quoi (spip08).

Les squelettes ont-ils le statut de code source ? L'analyse des entrevues ne permet pas de faire ressortir une réponse claire à cette question. Cette ambiguïté est probablement due en partie au fait que, méthodologiquement, nos entrevues étant semi-dirigées, nous n'avons pas posé exactement les mêmes questions à chaque personne, et que notre grille d'entrevue a évolué dans le temps. Notons tout d'abord que plusieurs des acteurs de SPIP sont réticents à considérer les squelettes de SPIP, ou le « langage de squelettes », comme étant de la « programmation », voire même du « code informatique ». Cette réticence s'expliquerait par le fait que les squelettes de SPIP servent avant tout à « structurer » l'information (ou à la mettre en forme) plutôt que de la manipuler. Dans ce sens, le langage de squelettes ne constitue pas un langage de programmation, comme PHP, mais s'assimile plutôt à un langage de balises, comme HTML :

> Est-ce que c'est là aussi de la programmation ? C'est pas très clair, c'est intermédiaire entre un langage de balisage standard comme HTML et du PHP simplifié. La frontière n'est pas claire (spip03).

> C'est pas du code informatique dans le sens où c'est... [pense] SPIP, c'est un langage qui permet de structurer l'information. Un langage informatique de façon plus générale, c'est fait pour faire pleins de manipulation. Et ce n'est pas fait pour travailler, nativement, sur la sortie de l'information, tu vois (spip09).

Ces propos ne disent cependant rien sur le statut des squelettes comme *code source*. D'autres participants à notre recherche sont cependant plus affirmatifs concernant ce statut des squelettes comme code source :

> *Q- Le code source de SPIP, qu'est-ce que ça désigne pour toi ?*
> Pour moi ça désigne, euh... [réfléchit] En premier, on va mettre le code PHP parce que c'est l'ossature, mais ça inclut aussi les squelettes SPIP (spip11).
>
> *Q- Tu dirais que les squelettes font partie du code source ?*
> Ah oui! (spip07).

Mentionnons que les squelettes – et le langage SPIP en général – sont à notre avis l'un des aspects les plus intéressants du projet SPIP car ils permettent à des personnes aux horizons disciplinaires différents (c'est-à-dire : pas seulement des gens qui ont été formés en informatique) de participer à la fabrication collective du logiciel. La spécification du langage de squelettes est l'objet de controverses au sein de SPIP, et à l'extérieur de la communauté. D'une part, parce qu'il s'agit d'un langage basé sur le français, ce qui coupe SPIP de la communauté extérieure et d'autre part, parce que le langage de squelettes ne correspond à aucun standard. Nous analyserons au chapitre 5 une controverse qui a fait surface à la suite d'une proposition de réforme du langage SPIP par l'un des acteurs, proposition de réforme qui a été perçue par plusieurs acteurs comme trop complexe pour l'utilisateur visé par SPIP.

Fichiers de configuration YAML

Les squelettes étant une particularité de SPIP, on ne les retrouve pas dans symfony. On retrouve toutefois dans ce dernier projet certains fichiers écrits dans le langage YAML, qui pourraient avoir un statut similaire à celui des squelettes de SPIP. D'ailleurs, comme nous le verrons au chapitre suivant, certains débats concernant l'accessibilité du code source ont pour objet le langage de squelettes dans SPIP, et les fichiers de configuration YAML dans symfony.

Selon l'un des participants à la recherche, le YAML est un langage de description des données, comme l'est par exemple le XML (*eXtended Markup Language*), un langage aujourd'hui très largement utilisé, par exemple, dans la description des fils de syndication RSS, ou encore, comme langage de base du standard *OpenDocument*, utilisé dans la sauvegarde des documents *OpenOffice* (comme cette thèse par exemple). Le YAML se distingue toutefois du XML par sa facilité de lecture et d'écriture pour l'humain, comme le

décrit cet extrait d'entretien par ailleurs très éclairant dans sa description des enjeux de lecture et d'écriture dans les relations humain-machine :

> Le XML, c'est facile à lire et écrire pour une machine, en revanche pour un humain c'est compliqué. Il faut ouvrir des balises, il faut mettre des guillemets, respecter une imbrication compliquée. [...] Le YAML c'est le contraire. C'est lisible et écrivable par des hommes, la machine, elle se débrouille, il y a des bibliothèques capables de transformer du YAML en structures de données PHP, XML, etc (sf03).

Le YAML est-il considéré comme du code source par les acteurs de symfony ? En suivant les propos de notre actrice pour qui le code source de symfony est « celui que je peux télécharger » (sf05), il faudrait répondre par l'affirmative. Le tableau 4.1 montre en effet que, dans le cas de symfony, l'archive téléchargée compte 157 fichiers portant l'extension .yml, signifiant qu'ils sont en format YAML. Lorsqu'interrogée spécifiquement sur ce format, cette même actrice ne considère cependant pas ces fichiers comme faisant partie du code source, les considérant plutôt comme les « *inputs* » de celui-ci.

> *Q- Des YAML... Ça, tu considèrerais que ça fait partie du code source? Parce que ça ne contient pas des boucles...*
> Ah! C'est une bonne question... C'est une très bonne question! Oui, selon ma définition, c'est assez naze, parce que ça ne rentre pas dans les définitions. Pour moi, ça configure le code source, mais ça ne fait pas partie du code lui-même. Parce qu'en fait, ces fichiers, finalement, dans symfony, dans le fonctionnement, ils sont lus, et après, ils sont transférés dans une mémoire... PHP, dans un fichier PHP, pour être réutilisés. Ils sont transférés finalement dans des variables accessibles en PHP, dans le système symfony, donc pour moi, c'est les fichiers qu'on donne à manger au début, en entrée – c'est les *inputs*, finalement, du code source (sf05).

Cette contradiction montre bien les difficultés à définir de façon précise le code source. Ce « flou définitionnel », qui apparaîtra davantage dans la suite du chapitre, est l'une des conclusions principales de notre thèse.

4.1.4 Le dépôt Subversion, ou l'organisation temporelle du code source

Au début de la section 4.1.1, nous avions suivi les propos d'une actrice pour qui « Le code source [...] c'est celui que je peux télécharger » (sf05). Nous avions alors téléchargé, puis analysé une archive en format .zip. Les pages de téléchargement de chacun des projets font cependant référence à une autre façon de télécharger le code source. Dans le cas de symfony,

la page mentionne qu'il est possible de retirer le code source depuis le dépôt Subversion (« *Checkout from the Subversion repository* » – figure 4.1), tandis que la page de téléchargement de SPIP indique quant à elle que SPIP est développé sous SVN (figure 4.2). Le *dépôt* ou le *répertoire* Subversion, aussi appelé SVN, constitue par conséquent un site important à analyser en ce qui concerne le code source. L'un des acteurs de SPIP soutient d'ailleurs que « tout ce qui est dans le répertoire SVN est code source » (spip11).

Dans les projets étudiés, comme dans beaucoup d'autres projets de développement de logiciels, la fabrication collective du code source s'appuie sur des dispositifs logiciels connus sous le terme de *systèmes de gestion des versions*. Ces logiciels permettent de conserver l'historique de toutes les modifications qui ont été effectuées sur un fichier ou un ensemble de fichiers. Dans les deux projets étudiés, le logiciel utilisé se nomme *Subversion*, souvent désigné simplement par l'acronyme *SVN*[88]. Subversion est d'ailleurs utilisé dans un grand nombre de projets, en particulier dans les projets de logiciels libres et open source. Par exemple, le site ohloh.net, que nous abordons plus loin, recense plus de 238 000 projets de logiciels libres ou open source, dont plus de 70% utilisent différentes variantes du système *Subversion*[89].

L'organisation du code source dans Subversion pourrait être décrite selon deux axes (notre catégorisation) : l'un qui pourrait être qualifié de « spatial » et l'autre plus temporel. Sur l'axe spatial, comme le montre la figure 4.6, le code source est organisé selon la notion générale de répertoire. Nous l'avons vu plus tôt dans le chapitre, le code source, pour chacun des projets, est constitué d'une multitude de fichiers et répertoires (voir le tableau 4.1). Cette organisation du code source est en fait une réplique exacte de son organisation dans le dépôt Subversion.

---

88 Cette situation a cependant évolué durant l'histoire des projets. Ainsi, SPIP utilisait de 2002 à 2005 un autre système similaire appelé CVS (*Concurrent Versions System*), avant d'utiliser Subversion. Durant la période de notre étude, les deux projets se sont également tournés vers un nouveau système de gestion des versions, nommé GIT, que nous présentons au chapitre 6.

89 <http://www.ohloh.net/repositories/compare> (consulté le 12 novembre 2010).

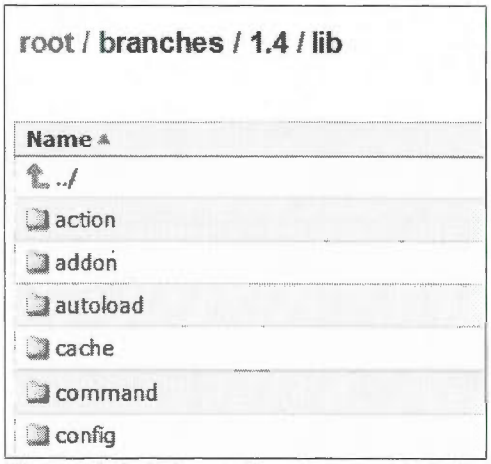


Figure 4.6 : Répertoires dans le dépôt Subversion
<http://trac.symfony-project.org/browser/branches/1.4/lib> (consulté le 31 janvier 2012).

Sur l'axe « temporel », les répertoires sont également utilisés pour organiser le code source selon ses différentes versions. Ces répertoires sont souvent désignés comme des branches, ou bien des tags. Ainsi, l'image suivante (figure 4.7) montre l'organisation du dépôt du code source, en différentes branches et tags, pour les cas de SPIP et symfony :

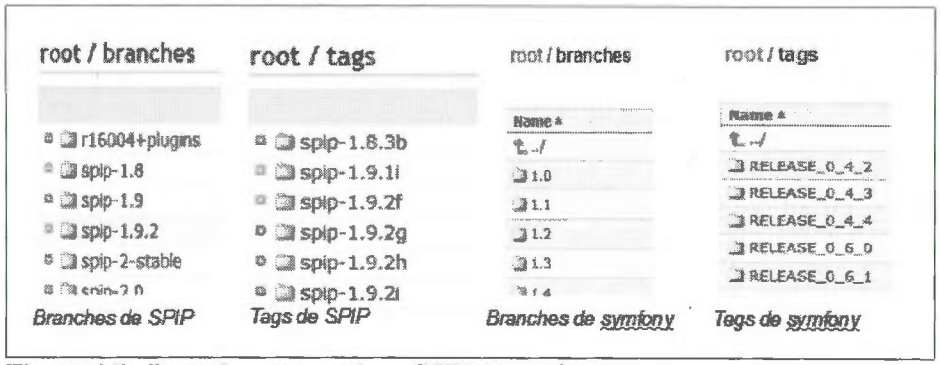


Figure 4.7 : Branches et tags dans SPIP et symfony
<http://trac.symfony-project.org/browser> et
<http://core.spip.org/projects/spip/repository/show> (consultés le 11 novembre 2011).

Dans les deux projets étudiés, les branches et les tags sont utilisés d'une manière similaire, dans l'objectif de marquer les différentes versions du code. Ainsi, comme le montre la figure précédente (4.7), le code pour chacun des projets est divisé entre différentes branches, qui constituent en quelque sorte des grandes versions du code des projets, par exemple 1.8, 1.9 ou

2.0 dans le cas de SPIP; 1.0, 1.1., 1.2, 1.3 dans le cas de symfony. Les tags sont quant à eux utilisés pour les versions mineures, par exemple, 1.9.2 pour SPIP ou encore 0.6.1 pour symfony, qui corrigent notamment certains bugs[90]. En particulier, les branches contiennent du code source susceptible d'être modifié, ou qui est toujours en développement, tandis que les tags constituent plutôt des « captures » du code source, à un moment donné. Par exemple, les responsables du projet décideront de publier la version 1.3 de symfony et ils créeront alors une nouvelle branche destinée à publier la version 1.4. Ainsi, la branche 1.4 sera utilisée pour tous les nouveaux développements, tandis que des *commits* pourront être faits sur la branche 1.3 afin de résoudre des bugs seulement. Un acteur de SPIP explique ainsi l'organisation du code source dans les différentes branches :

> Donc c'est ça l'organisation. A priori, 1.8 et 1.9, on ne les modifie que lorsqu'il y a des trous de sécurité avérés. Donc on bouche le trou de sécurité dans ces branches là, et on fait des *patches* de sécurité. La branche 2.0 stable, elle évolue encore un peu, des fois on rajoute des fonctions dedans, notamment pour des questions de compatibilité, si il y a un plugin important qu'on développe, et qu'il y a besoin d'un truc, qui soit dans le coeur, ben on va le mettre, on va encore modifier le coeur dans la version stable. Et puis, le 2.1, l'idée, c'est qu'on est en train de la découper en morceaux. C'est comme la 2.0, mais on la découpe en morceaux, on prend des tas de fonctionnalités et on les enlève du coeur. On les transforment en plugins (spip01).

L'organisation temporelle du code source dans Subversion se fait également autour des notions de *révision* et de *commit,* aspects qui seront abordés au chapitre 6. Pour le moment, disons simplement que le commit est une commande informatique qui consiste à envoyer ses modifications locales du code source vers le serveur central. La révision constitue quant à elle la version spécifique du code source associée à ce commit. Ainsi, le code source déposé dans Subversion pourra être l'objet de plusieurs modifications successives qui seront chacune associées à un numéro de révision (voir figure 4.8). L'analyse du processus conduisant à ces modifications fera l'objet du chapitre 6.

---

90 La distinction entre branches et tags est surtout conventionnelle. En effet, le logiciel Subversion ne distingue pas ces deux commandes qui peuvent être réalisées par la commande *cp* (copy). Cependant, ces deux commandes étaient distinctes dans le logiciel CVS, qui est en quelque sorte l'ancêtre de SVN.

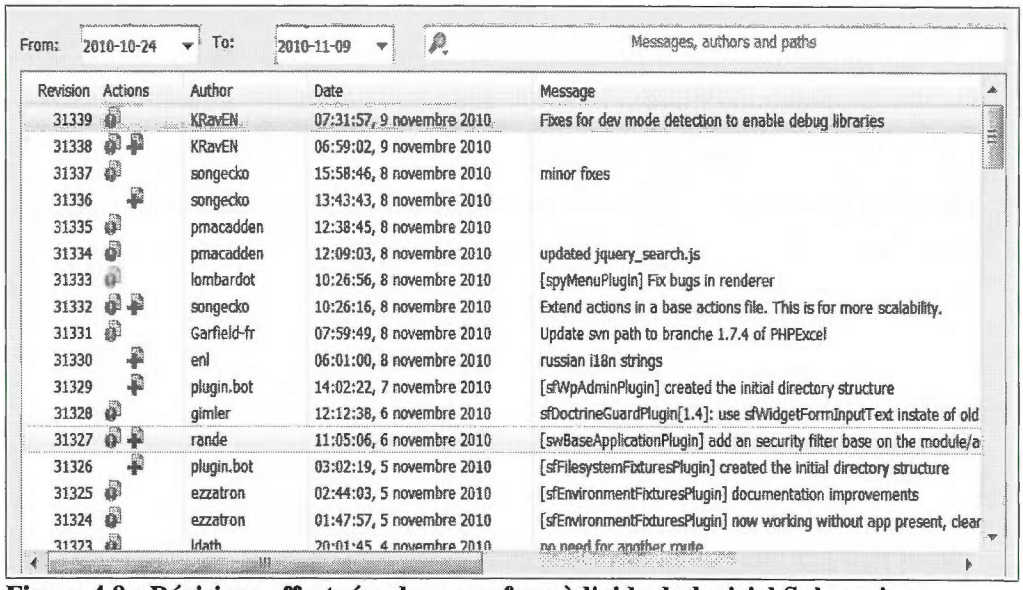


Figure 4.8 : Révisions effectuées dans symfony à l'aide du logiciel Subversion
Obtenu à l'aide du logiciel TortoiseSVN en explorant le dépôt <http://svn.symfony-project.com>.

Au niveau de l'organisation « spatiale », mentionnons également que le code source déposé dans Subversion n'est pas uniquement celui du « *core* ». Le dépôt Subversion contient plusieurs autres répertoires tels que *doc* et *plugin* qui contiennent respectivement la documentation technique de symfony, ainsi que les plugins. Dans SPIP, un autre dépôt Subversion a été créé, appelé *SPIP-Zone*, dans lequel sont notamment déposés les plugins, les squelettes et la documentation.

Mentionnons finalement que, bien qu'utilisés principalement dans le monde de l'informatique pour la gestion des différentes versions du code source, les systèmes de gestion des versions peuvent en fait gérer tout type de documents en format texte, voire même, si on accepte de ne pas utiliser certaines fonctionnalités, tout type de document numérique. Dans symfony, et dans une moindre mesure dans SPIP, ce système est par exemple utilisé pour faciliter l'écriture de la documentation technique ou encore les guides d'introduction à l'usage de symfony. Cette remarque nous amène à considérer la manière dont notre analyse du code source peut être généralisable à d'autres types de documentation, comme nous l'aborderons en conclusion de la thèse.

4.1.5 Autour du code source : « *Code snippets* », noisettes et autres morceaux de code source

> *Q- Et les code snippets?*
> C'est des bouts de code, pour répondre à des problématiques particulières. Les *snippets* qu'il y a sur symfony, nous on les a aussi en interne pour travailler (sf04).

Autour de ces éléments de code source que nous avons mentionnés plus tôt, on retrouve à plusieurs endroits des artefacts décrits par les acteurs comme des « bouts de code » ou des « morceaux de code ». L'importance de ces morceaux de code s'explique en bonne partie par les pratiques d'écriture qui, favorisées en cela par les licences de logiciels libres, s'appuient largement sur le copier-coller :

> Il faut bien voir que la façon de coder, enfin c'est aussi ma façon de coder, je ne tape pas beaucoup de code. C'est énormément de copier-coller. De récupération de bouts, parce que la structure, les fautes de frappe et tout, on en enlève pas mal quand on récupère des bouts entiers (sf04).

Dans le cadre des projets étudiés, certains de ces bouts de code sont davantage formalisés. Dans symfony, on retrouve par exemple un répertoire public de *code snippets* (figure 4.9, page suivante), d'une manière qui rappelle la forme « réseau social », où ces morceaux de code peuvent être partagés, commentés et catégorisés par des tags, et synchronisés par des fils RSS[91]. Bien que nous n'explorerons pas davantage ces « *code snippets* » dans le cadre de notre thèse, mentionnons ici que cette manière d'encapsuler des morceaux de code source et de les partager dans la forme « réseau social » est l'un des aspects qui nous semble le plus intéressant, en regard de la circulation et du partage du code source dans les projets étudiés.

Dans SPIP, on retrouve une logique similaire sous l'appellation de *noisettes*. Le terme noisette renvoie ici à l'« imaginaire » de l'écureuil qui s'exprime principalement dans le choix du terme SPIP comme nom du projet, faisant référence au personnage de l'écureuil dans la bande dessinée Spirou) et, par conséquent, dans le choix du logo. Selon un acteur, les noisettes sont « des petits morceaux de squelettes qui représentent des morceaux de pages, soit de l'espace privé, soit l'affichage de formulaire, etc. » (spip01). Un projet appelé « noisetier », en

---

91 Par ailleurs, notons que l'une des choses remarquables de ce répertoire de « *code snippets* », est qu'il s'agit d'un des seuls endroits dans les deux projets étudiés, où l'expression « code source » (en anglais : *source code*) est explicitement utilisée.

chantier au moment de notre enquête, permet de regrouper ces différentes noisettes ou morceaux de squelettes.

Figure 4.9 : Le répertoire de « code snippets » de symfony
<http://snippets.symfony-project.org/> (consulté le 18 mars 2011).

4.2 Définitions formelles et métaphores du code source dans les entrevues

Dans le chapitre de problématique, nous avons constaté que la définition de la notion de « code source » ne faisait pas l'unanimité dans les écrits. Ainsi, Wikipédia donne une définition assez précise du code source comme un « ensemble d'instructions écrites dans un langage de programmation informatique [...][92] ». L'*amicus curiae,* que nous avons cité plus tôt, définit le code source de façon beaucoup plus générale comme un « sous-ensemble de l'expression humaine que les ordinateurs peuvent interpréter et exécuter[93] ». Rappelons également que les auteurs de ce document soutiennent que la définition de la notion de code source change rapidement, au fur et à mesure que les technologies numériques se développent.

92 <http://fr.wikipedia.org/w/index.php?title=Code_source&oldid=52775534> (consulté le 6 juillet 2010).

93 <http://cryptome.org/mpaa-v-2600-bac.htm> (consulté le 17 février 2011).

Cette seconde partie du chapitre fait écho à cette constatation préliminaire d'une définition ambiguë du code source dans les écrits, mais en nous appuyant cette fois-ci sur notre matériel empirique. D'un point de vue méthodologique, notre approche dans ce chapitre s'appuie largement sur l'analyse des entrevues et consiste principalement à « suivre » les différentes définitions que les acteurs donnent au code source, en juxtaposant certains extraits d'entrevues pour faire ressortir les nuances et les contradictions dans ces définitions. Nous explorons également certaines métaphores utilisées par les acteurs pour décrire le code source, en tant qu'elles constituent des comparaisons qui permettent de saisir la manière dont les acteurs perçoivent la réalité du code source. Rappelons que, selon Proulx (qui cite sur le dictionnaire Larousse), la métaphore se définit comme le « procédé par lequel on transporte la signification propre d'un mot à une autre signification qui ne lui convient qu'en vertu d'une comparaison sous-entendue » (Proulx, 2007c, p. 113). Proulx, à la suite de Krippendorf, suggère ainsi de prendre au sérieux les métaphores des acteurs, car elles organisent les expériences de communication des acteurs et peuvent par conséquent participer à créer des réalités (Krippendorff, 1993, p. 5; cité par Proulx, 2007c, p. 113) .

Cette seconde partie du chapitre fait notamment ressortir certaines ambiguïtés concernant le caractère écrit du code source, de même que son caractère relationnel. L'analyse de ces définitions met également de l'avant la possibilité d'une définition extensive du code source.

4.2.1 Le code source, un écrit ?

Le premier aspect qu'il nous semble important d'aborder concerne le présumé caractère *écrit* du code source. Nous avons en effet vu en problématique la manière dont le caractère écrit du code source est souvent mis de l'avant dans certaines définitions et par certains auteurs. Comme mentionné plus tôt, Wikipédia définit le code source comme un ensemble d'instructions « écrites ». Dans son analyse du pouvoir régulateur du code informatique, Lessig (2006) fait référence pour sa part aux « code writers » comme des législateurs. De la même manière Brousseau et Moatty (2003) parlent quant à eux de « l'écriture de lignes de code informatique » (p. 8) dans leur ouvrage. Cependant, nous avons également constaté dans notre recension des quelques définitions formelles (section 1.3), que le code source n'est pas toujours défini comme un écrit. Mentionnons par exemple de nouveau le site web des

conférences SCAM, qui considère que le code source peut notamment inclure des représentations graphiques[94].

La perspective des acteurs dans les projets étudiés est assez similaire. La plupart des participants à nos entrevues parlent du code source comme quelque chose « qu'on écrit ». Certains des acteurs le définissent d'ailleurs comme une suite d'instructions :

> Du point de vue des développeurs, ce qu'on appelle le code source c'est justement la rédaction même de ces instructions dans un langage particulier (spip07).
>
> Et, pour donner des instructions à une machine, il faut écrire des programmes dans un langage qui est compréhensible par des humains. Ces instructions, ça va correspondre au code source. [...] Le code source, c'est vraiment un *listing*, du texte (sf06).
>
> Dans l'open source, c'est la même chose, sauf qu'au lieu d'écrire du texte, on écrit du code. Et c'est lu par des gens qui sont des développeurs (sf03).

Cette façon de considérer le code (source) comme quelque chose qu'« on écrit » est particulièrement prégnante dans les échanges avec les acteurs, au point où, pour un certain temps, nous souhaitions orienter l'aspect central de notre thèse sur cette propriété du code source comme une forme d'écriture[95]. Toutefois, si le caractère écrit du code source semble généralement accepté dans le langage commun des acteurs, il reste néanmoins problématique concernant une définition plus formelle du code source. Ainsi, lorsqu'interrogé plus formellement sur la définition précise qu'il donnerait au code source, un acteur apporte certaines nuances, en notant par exemple que dans le code source, il y a la documentation :

> [...] Le code source, c'est vraiment un *listing*, du texte.
> *C'est juste du texte ? Le code source c'est du texte ?*
> C'est très résumé. Le code source bien sûr, ce n'est pas que du texte, mais dans le cadre de cette question, bien sûr, s'il fallait que j'explique à un novice ce qu'est le code source, je dirais que oui, le code source c'est du texte, qui correspond à un certain nombre d'instructions. Sachant qu'après le code source, il y a plusieurs choses. Il y a la documentation, il y a plusieurs choses derrière (sf06).

Un autre acteur de SPIP relate pour sa part son expérience passée dans une entreprise de génie mécanique et soutient que le code source peut recouvrir différentes formes :

---

94 <http://cist.buct.edu.cn/zheng/scam/2011/indexse13.html> (consulté le 28 février 2012).

95 C'est d'ailleurs ce qui nous a amenés à explorer les travaux d'anthropologie de l'écriture.

> Nous on faisait la spécification sous forme de schémas-blocs qui s'appellent *matlab simulink* qui servent à simuler des systèmes, etc. [...], mais dans un projet comme ça, on peut considérer que les schémas-blocs qui ont servi à spécifier et qui sont exécutés dans un environnement donné, sont aussi une forme de code source (spip11).

Dans le cas mentionné par cet acteur, le code source prend la forme de schémas-blocs, c'est-à-dire, une forme *graphique*, plutôt qu'écrite, qui a pour fonction de simuler un système informatique. Une fois la simulation mise au point, le modèle en schémas-blocs est traduit dans un autre langage informatique par des acteurs humains[96]. On retrouve à la figure 4.10 un exemple de schémas-blocs.

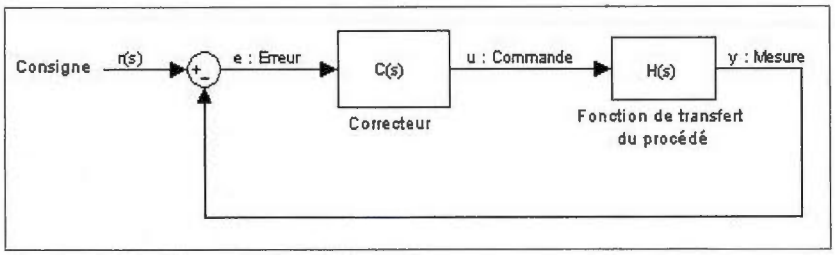


Figure 4.10 : Exemple de schémas-blocs
<http://fr.wikipedia.org/wiki/Schéma_fonctionnel> (consulté le 9 mai 2012).

Un autre acteur de SPIP – celui-là même qui avait initialement soutenu que la notion de code source n'avait pas de sens dans le monde PHP[97] – va dans le même sens en considérant que le code source peut prendre la forme de maquettes graphiques, qui agissent à titre de prototype fonctionnel :

---

96 Cette description met l'accent sur deux points cruciaux : le code source n'est pas toujours un artefact écrit, et l'opération de traduction n'est pas toujours automatisée, pouvant être le fait d'un humain.

97 Rappelons que pour cet acteur (spip01), la notion de « code source » ne fait pas sens pour le code écrit en langage PHP, puisque ce code est exécuté directement par la machine (ou plutôt : interprété). Il ne s'agirait donc pas de code source puisqu'il n'y a pas, dans ce cas-ci, d'opération de compilation permettant de traduire le code source en code objet (voir le chapitre 3, p.113; nous avons également montré à ce moment que la notion de code source était cependant bel et bien mobilisée dans le vocabulaire des acteurs).

> Mais [elle, une actrice] fait des prototypes fonctionnels. Et ces prototypes fonctionnels, une fois qu'ils sont conçus, et une fois qu'elle nous montre sur l'écran, avec une espèce de maquette, que si on clique là, ça fait ça, si on clique là, ça fait ça, etc. [...] La source programmatique, elle est dans la maquette fonctionnelle. Donc, c'est ça qui a le statut de code source (spip01).

Dans cette partie de l'entrevue, notre interlocuteur abordait alors une problématique politique qu'il constatait, à savoir que certaines personnes au sein de SPIP déprécient leurs contributions dans le projet, car elles ne sont pas perçues comme des contributions sous forme de code. Nous avons nous-mêmes constaté une telle relation ambiguë avec la catégorie du code – et celle de codeur – à plusieurs moments dans notre recherche, un aspect que nous abordons plus loin dans ce chapitre (section 4.3)[98].

Pour cet acteur de SPIP que nous venons de citer, le code source d'un logiciel constitue non pas tant ses instructions écrites, mais plutôt le lieu où les spécifications du fonctionnement du logiciel sont formulées : « Le code source, le vrai code source, c'est ta spécification » (spip01). Ainsi, les prototypes fonctionnels sous forme de maquettes graphiques constituent une sorte de spécification du fonctionnement du logiciel et, à ce titre, agissent comme du code source. Ainsi, dans cette logique, le code source de l'Internet correspondrait pour cet acteur aux *Request For Comments* (RFC) :

> Ou en ce qui concerne Internet, c'est surtout des RFC.
> *Q- Les RFC ce serait le code source ?*
> Ben, le code source d'Internet. Ben oui, c'est clair. Parce que tu peux tout éliminer et le remplacer par autre chose, du moment qu'ils respectent les RFC, qu'ils respectent les normes (spip01)[99].

4.2.2 Le code source : le code de référence, celui qu'on va « pétrir »

Ce dernier extrait d'entrevue, qui associe les RFC au code source de l'Internet, renvoie à une définition beaucoup plus générale du code source. Est code source ce qui constitue la

---

98 Mentionnons toutefois que la réponse de notre interlocuteur à cette problématique, qui consiste à généraliser la définition du code source pour inclure un spectre plus large d'artefacts et d'activités, nous semble l'inverse de celle mise de l'avant dans certains travaux féministes qui préconisent de valoriser les activités non liées à la production de code (Lin, 2006b; Haralanova, 2010).

99 Les *Request For Comments* (RFC) sont des documents qui décrivent certains aspects de l'Internet. Initiés par des experts et élaborés conjointement par la communauté, certains de ces RFC, mais pas la totalité, constituent des standards de l'Internet. Pour une analyse de la « culture de la gratuité » dans la réalisation collective des RFC, voir Proulx et Goldenberg (2010)

spécification de base d'un logiciel, d'un dispositif informatique, voire même de l'Internet. Notons par ailleurs que cette définition n'implique pas que la spécification en question soit directement comprise par la machine, qu'elle soit exécutable. Dans le cadre de cette entrevue, dont l'analyse nous a causé beaucoup de maux de tête, nous avons ensuite amené notre interlocuteur à préciser cette définition du code source :

> *Q- Donc, toi, tu qualifierais le code source, un peu comme le lieu où la créativité se fait ? Le lieu de l'invention finalement ?*
> Euh, de l'invention, pas forcément. Mais c'est le lieu où se fait le travail de définition précis des fonctionnalités. C'est pas en conception large, etc., qui dit que ce serait bien si on avait un logiciel pour gérer les articles. Ça non. Mais c'est un truc précis qui dit que j'ai tel bouton, de telle forme, etc. Et quand je clique dessus, ça provoque tel élément, ça fait appel à tel algorithme (spip01).

Ainsi, le code source serait le lieu précis où sont établies les spécifications du logiciel.

Pour plusieurs participants, une caractéristique essentielle du code source serait sa dimension « originale » ou initiale :

> Au final, le code reste du code source, parce que c'est lui qui est à l'origine du fonctionnement d'une application (sf07)

Cet acteur compare le code source à une recette de cuisine qui décrirait les ingrédients et les « instructions » pour fabriquer un plat. La recette permet donc de fabriquer le plat, mais ne constitue pas le plat lui-même.

> Ces instructions, ça va correspondre au code source, ça va être un petit peu la recette de cuisine du programme, et puis une fois que l'ordinateur va exécuter ces instructions, et bien, on va avoir un programme qui va être le gâteau, en quelque sorte, qui est le produit fini (sf06).

La métaphore de la recette est intéressante d'une part, car elle permet de mettre l'accent sur un certain caractère écrit du code source (la recette étant composée d'instructions écrites). Elle met d'autre part de l'avant les deux modes d'existence d'un programme, comme code source et comme code exécutable.

La prochaine citation met clairement de l'avant la dimension relationnelle du code source et son rapport à une opération de traduction.

> Ça c'est un squelette. C'est du HTML, à peu de choses près, il y a du SPIP dedans. Tu vois, ça c'est du SPIP. Et ça c'est du HTML. [...] Quand SPIP traite ce code source qui est du HTML-SPIP, il produit un code objet. [...]. Ça c'est un code PHP. Mais ça fait pas partie du logiciel. Donc, c'est le résultat de la compilation, par le compilateur de SPIP, d'un squelette source. Donc, ça c'est du code objet si tu veux, c'est du PHP. Mais le code source de ce squelette, c'est du HTML, c'est des boucles SPIP (spip01).

Dans SPIP, les squelettes sont intimement liés au concept de compilateur, traditionnellement au fondement de la notion de code source. Ainsi, les squelettes sont écrits dans un langage HTML-SPIP, puis sont compilés, c'est-à-dire traduits en un autre langage qui est du PHP. Donc, comme cet acteur le mentionne, les squelettes constituent dans ce cas-ci le code source, qui sera compilé, pour devenir un code objet écrit en PHP (qui pourrait ensuite être considéré comme le code source de la page HTML).

Pour un autre acteur, le code source est en quelque sorte l'ADN, la « matrice initiale humainement compréhensible » (écrite ou non), le code de référence duquel découlent les autres, et qu'on va *pétrir* pour faire évoluer le logiciel :

> Par rapport au code source, qui lui est la matrice, l'ADN, la matrice initiale, la version initiale humainement compréhensible modifiable, reproductible, en fait retravaillé [...] En fait, le code source je pense c'est aussi celui, sur lequel on va travailler, qui est malléable, et qu'on va faire évoluer, qu'on va pétrir pour faire évoluer le logiciel final [...] C'est celui qui est utilisé comme référence pour produire les autres codes qui vont servir, et celui qui est à la base, enfin, qu'on fait évoluer pour faire évoluer le logiciel. Et duquel découlent les autres en fait (spip11).

Cette manière de définir le code source comme le code « qu'on va pétrir » et qui est utilisé « pour produire les autres codes » est sans doute l'une des définitions les plus formelles et générales du code source qui nous a été donnée. Cette définition est d'ailleurs assez similaire à celle explicitée par la licence publique générale GNU, que nous avons citée en problématique, soit « the preferred form of the work for making modifications to it[100] ».

---

100 Notons que Richard Stallman ne serait sans doute pas d'accord avec la comparaison que fait cet acteur entre le code source et l'ADN. Celui-ci écrit ailleurs que « Natural organisms never had anything like source code. The genetic code of an organism is more comparable to a binary (in fact, quaternary) executable ».
<http://blog.reddit.com/2010/07/rms-ama.html> (consulté le 24 février 2012).

4.2.3 « Code source de code source de code source ». Le code source dans une chaîne de traduction

Notons que certaines définitions du code source ne renvoient pas nécessairement à l'artefact d'origine, « celui que l'on va pétrir ». Plusieurs acteurs insistent sur la dimension relationnelle du code source, à savoir que le code source peut aussi être le résultat d'un autre code source. Certains extraits d'entrevue montrent bien cette dimension relationnelle :

> Donc, il y a déjà cette dimension de code source de code source de code source. Après, si tu remontes, tu arrives forcément à un moment où tu es à l'assembleur (sf05).

> Après, il y a différents niveaux de code source en fonction de ce qu'on utilise. Pour PHP, il y aura le code source de PHP, qui lui repose sur un code source C. Et puis par-dessus PHP, on aura une couche symfony, le code source, puis le code source de l'application qu'on développe avec symfony (sf07).

Dans l'analyse des conversations par courriel, il apparaît que le terme code source fait parfois référence au code source d'une page HTML ou d'un courriel. Dans un échange sur une liste de discussion, un des acteurs parle ainsi du « code source du courriel que je reçois » (spip-zone). La figure 4.11 montre comment l'on peut accéder au code source d'un courriel dans Thunderbird.

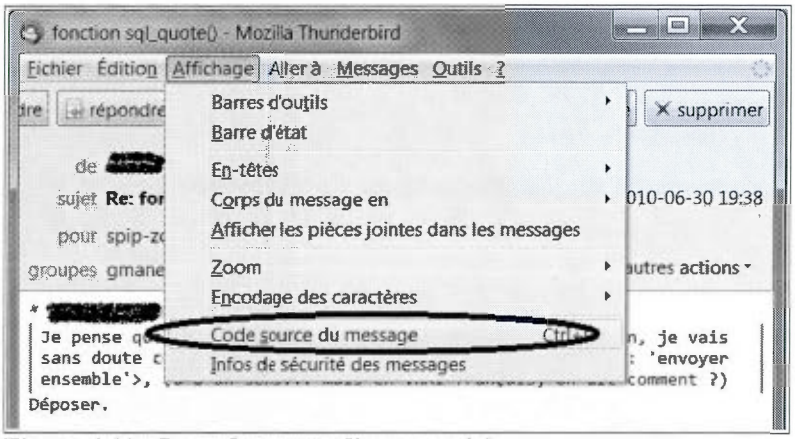


Figure 4.11 : Le code source d'un courriel
(image réalisée à partir d'une capture d'écran de Thunderbird).

Dans ce cas-ci, le code source du courriel ne constitue pas le code « qu'on va pétrir » car il est plutôt le résultat d'une opération de traduction automatique réalisée par le logiciel de courriel électronique. Cependant, il constitue également le code source de référence, le code source *du* courriel. On voit ici la dimension relationnelle du code source, qui se situe toujours dans

une chaîne de traduction, soit à l'origine ou dans une position intermédiaire, dans lequel cas il s'agit d'un code généré qui devient lui-même le code source d'un niveau de traduction supérieur.

L'un des aspects qui semble fondamental dans la définition du code source concerne cette dimension relationnelle. Ainsi, dans l'ensemble de nos entrevues, les acteurs définissent toujours le code source par rapport à quelque chose d'autre : « le code source d'Internet », « la recette du gâteau », « celui qui est utilisé comme référence », « celui qui est à l'origine du fonctionnement d'une application » et finalement, dans notre dernière capture d'écran (figure 4.11) : « le code source *du* message ». Le code source est en effet toujours la source de quelque chose d'autre, d'un logiciel, d'une application web, d'un autre code. Ainsi, traditionnellement, le code source est mis en relation avec le code exécutable, par l'intermédiaire d'un compilateur. On pouvait alors dire que le compilateur traduisait le code source en code exécutable. Le code source était le code source du code *exécutable* et éventuellement du logiciel, lorsque ce code exécutable était effectivement exécuté par l'ordinateur. Cependant, comme nous l'avons mentionné, cette relation « traditionnelle » entre code source et code exécutable est problématique dans le cadre de logiciels, comme SPIP et symfony, écrits dans des langages interprétés tels que PHP, et où la notion de compilateur est absente. C'est d'ailleurs précisément le fait que le code (source) de SPIP soit interprété, et non pas compilé, qui a amené notre interlocuteur à affirmer que la notion de «code source » ne faisait pas sens dans SPIP, même si plus tard, celui-ci en est venu à affirmer que les RFC pouvaient constituer en quelque sorte le code source de l'Internet.

Nous nous intéressons principalement ici à cet aspect du code source, en tant qu'il constitue l'artefact *à l'origine de*. Cependant, notre regard se centre surtout sur le code source fabriqué par les humains, « celui que l'on pétrit », et non pas le code source informatiquement généré, comme dans le cas du code source d'un message de courriel.

4.2.4 Le statut de la documentation et des commentaires

L'un des traits marquants qui ressort de la plupart des entrevues, autant dans SPIP que dans symfony, concerne la relation entre le code source et la documentation. Cette relation est particulièrement importante pour plusieurs des acteurs qui vont jusqu'à affirmer qu'un code source qui n'est pas documenté n'existe pas :

> On a quelque chose qui pour nous est très important dans la communauté symfony c'est se dire qu'un code qui n'est pas documenté est un code qui n'existe pas (sf02).
>
> Alors c'est vrai qu'un logiciel qui n'est pas documenté, c'est un peu comme s'il n'existait pas. Si le code source n'est pas documenté, il n'est pas utilisable (sf06).
>
> La fonction en tant que telle ne peut pas exister sans la documentation et vice versa. [...] Parce qu'une fonction qui n'est pas expliquée ne sert à rien (spip10).

Dans cette dernière entrevue (spip10), ce participant nous expliquait qu'il arrive parfois qu'une partie de code (une fonction étant une partie du code) avait simplement été oubliée parce que celle-ci n'avait pas été documentée. Dans une phase subséquente de développement, du nouveau code a été fabriqué pour réaliser une fonction similaire à celle du code précédent, rendant pour ainsi dire *obsolète* le code précédent :

> Il y a des fonctions qui sont devenues obsolètes, parce qu'on ne sait pas à quoi elles servent, ou on a réinventé la roue en parallèle, on a réinventé une fonction similaire (spip10).

Ces derniers extraits d'entrevues montrent bien la manière dont le code source est articulé à d'autres artefacts et à un certain travail de maintenance. Bien sûr, concrètement, un morceau de code source existera toujours en tant qu'écrit, mais il ne sera pas utilisable, ou ne servira à rien. Nous pourrions dire qu'il n'existera plus en tant que code. Nous pouvons ici faire un lien avec les travaux en anthropologie de l'écriture qui s'appuient sur la théorie de l'acteur-réseau pour affirmer que la performativité des écrits dépend en bonne partie de leur *vivacité*. Ainsi, dans leur analyse du cas de la signalétique du métro de Paris, Denis et Pontille (2010a) montrent bien que la performativité des panneaux de signalétique dépend en bonne partie d'un travail de maintenance. Un artefact signalétique qui ne fait pas l'objet d'un entretien régulier risque que de ne plus être reconnu en tant que tel, et perdra par conséquent sa force performative. Les propos des acteurs nous amènent à une conclusion similaire, en ce sens que sans une documentation constamment mise à jour, le code source reste inutilisable et, dans ce cas, risque de sombrer dans l'oubli et devenir progressivement obsolète : il n'est plus performatif.

La documentation des projets étudiés prend généralement deux formes distinctes (notre catégorisation). D'une part, une documentation « externe » qui peut prendre la forme d'une page web documentant différentes parties du code source. Cette documentation « externe »

peut également prendre la forme d'un livre, en format papier, qui guidera le développeur dans son usage du code source[101]. D'autre part, la documentation peut prendre la forme d'une documentation « interne » qui correspond aux commentaires intégrés directement dans le code source. Ces commentaires sont généralement rédigés dans une langue naturelle, telle que le français ou l'anglais, et n'ont aucun impact sur le fonctionnement du logiciel : ils ne sont que des commentaires. La figure suivante (4.12) montre par exemple les commentaires dans un morceau de code source de symfony. Ces commentaires sont délimités par les symboles /** et */ (chaque ligne de commentaire est également précédée du symbole *, par soucis « esthétique », symbole qui n'a cependant aucun effet sur l'exécution du programme) :

```
29 class sfMySQLDatabase extends sfDatabase
30 {
31   /**
32    * Connects to the database.
33    *
34    * @throws <b>sfDatabaseException</b> If a connection could not be created
35    */
36   public function connect()
37   {
```

Figure 4.12 : Commentaires dans le code source de symfony
<http://trac.symfony-project.org/browser/branches/1.4/lib/database/sfMySQLDatabase.class.php> (consulté le 27 février 2012).

La ligne 32 de ce morceau de code source, où il est écrit « Connects to the database. », est typique d'un commentaire écrit dans une langue naturelle (l'anglais, dans ce cas-ci), et dont le but est d'aider à une meilleure compréhension du reste du code source. Pour certains acteurs, la documentation n'est pas seulement une aide à la compréhension du code source, mais peut elle-même avoir le statut de code source. On peut distinguer quatre manières dont la documentation a le statut de code source.

Premièrement, comme nous l'avons mentionné plus tôt, la documentation a le statut de code source dans la mesure où sans documentation, le code source n'existe pas.

---

101 Dans les deux projets étudiés, les manuels en format papier sont d'ailleurs souvent des reproductions exactes de la documentation en ligne. Voir par exemple <http://programmer.spip.org/Le-livre> (consulté le 27 février 2012).

Deuxièmement, la documentation revêt pour certains acteurs le statut de code source, dans la mesure où elle constitue le lieu de la spécification formelle du fonctionnement des logiciels. Cette position est celle tenue par notre interlocuteur (spip01), que nous avons amplement cité plus tôt, et qui mentionnait par exemple que les RFC constituaient le code source de l'Internet. Selon cet acteur « tu peux faire quelque chose de programmatique en faisant d'abord la documentation, et à ce moment, elle fait partie du code» (spip01). Bien que cette perspective ne soit pas aussi explicite dans les autres entrevues que nous avons réalisées, des acteurs de symfony font référence à une méthodologie de développement de logiciels, le *Documentation-Driven Development (DDD)*, où l'un des principes semble être d'expliquer d'abord, et de faire fonctionner ensuite (« explain first, make it work afterwards »[102]). Dans cette optique, le code source, écrit avec des instructions informatiques, constitue donc la résultante de la documentation.

Troisièmement, la documentation a le statut de code source, lorsqu'elle est directement intégrée à l'intérieur de ce qui est généralement considéré comme du code source. Nous faisons ici référence à cette catégorie de documentation que nous avons qualifiée d'« interne » et qui prend la forme de commentaires. La figure 4.12, présentée plus tôt, montre ainsi une intrication étroite entre commentaire, en langue naturel, et commentaire, en langage structuré. Nous avons en effet vu au tableau 4.1 que, dans le cas de symfony, ce que les acteurs désignent comme étant le code source de symfony comprend de nombreux fichiers en format .php dont plus du tiers des lignes de texte (ou de code) sont en fait des commentaires (110 773 lignes sur 302 803).

Quatrièmement, la documentation – et plus spécifiquement les commentaires – revêt le statut de code source dans le sens qu'elle contient en elle-même des codes permettant de générer une autre documentation de plus haut niveau. Plusieurs des projets de développement de logiciels s'appuient sur des dispositifs de création automatique de documentation. Dans le cas de symfony, c'est par exemple le logiciel PHPDocumentor[103]. La manière d'opérer de ces dispositifs de génération automatique de la documentation consiste à extraire certaines parties des commentaires pour créer des pages HTML. La ligne 34 du morceau de code source

---

102 <http://redotheweb.com/2008/09/23/document-driven-development-in-practice-rethinking-sfforms/> (consulté le 17 novembre 2011).

présenté à la figure 4.12 est un exemple d'un commentaire contenant des « codes » destinés à la génération automatique de la documentation. Cette information est un « commentaire en contexte d'exécution du code » (spip11, voir plus loin) mais dont le statut change dans d'autres contextes. Cette ligne de commentaire présente en effet une forme plus structurée, qui intègre certains symboles rappelant le code HTML : « @throws sfDatabaseException If a connection could not be created ». Le terme « @throw » constitue en fait une commande indiquant d'afficher ce texte en HTML. En d'autres termes, cette ligne de commentaire constitue le code source de la documentation générée automatiquement.

Cet extrait d'entrevue montre bien le rôle de ces commentaires qui changent de statut en fonction du contexte :

> Et donc, ce qui est commentaire dans un contexte d'exécution du code, va être une information complètement primordiale pour faire tourner les jeux de tests. [...] Il peut y avoir de l'information structurée dans les commentaires qui sont utilisés dans certains contextes par certains outils [...] (spip11).

Ce statut de certains commentaires, qui mélangent langage naturel et langage structuré (des « codes ») visant ensuite à générer un autre type de code, constitue un point de passage intéressant entre notre étude du code source principalement écrit en PHP, et d'autres formes d'écriture numériques. Nous présenterons par exemple en conclusion le cas des wikis dont le langage d'édition pourrait rejoindre celui de la documentation. L'analyse comparée de ces formes d'écriture pourrait être particulièrement pertinente dans la perspective d'une recherche plus générale sur les nouvelles formes d'écriture humain-machine. La problématique des commentaires est en effet ici similaire à celle de n'importe quel « code source » de mise en page, comme le code source d'une page HTML ou encore, celui d'une page Wikipédia. On a

---

103 Il est assez difficile de trouver cette information sur les sites web des projets respectifs. Dans symfony, on retrouve la mention de PHPDoc sur la page *How to contribute to symfony* <http://trac.symfony-project.org/wiki/HowToContributeToSymfony> (consulté le 24 septembre 2011). Dans le cas de SPIP, au moment de réaliser nos entrevues, un participant (spip10) mentionnait qu'« on a une volonté d'aller sur une écriture de type de prédocumentation PHPDoc », ce qui signifie qu'à ce moment (en avril 2010), cette pratique n'était pas instaurée. Certaines modifications plus récentes au code source de SPIP montrent toutefois que PHPDoc semble aujourd'hui utilisé.
<https://github.com/spip/SPIP/commit/5e525de455067db576b706a2733e01db44bd51bd>
(consulté le 22 mars 2011).

en effet ici affaire à un « code » de très haut niveau, lisible par l'humain, mais contenant également des symboles, ou des instructions écrites, destinés à un traitement informatique.

Mentionnons finalement que pour plusieurs acteurs, le code source lui-même peut agir à titre de documentation. Un des acteurs de symfony écrit par exemple qu'il lui arrive fréquemment d'aller voir directement dans le code source, pour comprendre comment ça fonctionne :

> Ça m'arrive fréquemment, quand j'ai besoin d'une fonctionnalité dans symfony, et puis que j'arrive à la limite de la documentation, et bien, d'aller voir directement dans le code, comment c'est fait, comment ça fonctionne » (sf06).

Cet acteur note toutefois que d'aller voir dans le code source ne permet pas de comprendre le fonctionnement global du logiciel, ce qui nécessite la documentation, mais seulement de mieux comprendre le fonctionnement de cas spécifiques :

> Dans des cas concrets, quand on a une certaine fonction et qu'on ne sait plus c'est quoi ses arguments, etc., là, c'est adapté d'aller voir dans le code source, parce que l'information est directement lisible en un coup d'œil » (sf06).

Ces aspects concernant la lisibilité du code source seront cependant abordés dans le chapitre suivant.

4.2.5 Métaphores spatiales et d'organisation du code source

Différentes métaphores mobilisées par les acteurs mettent l'accent sur l'idée d'une spatialité du code source, voire d'une certaine matérialité. La mention par plusieurs acteurs de l'idée de « descendre », ou d'aller « à l'intérieur » du code source, nous a interpellés à certains moments. Ainsi, lors d'une entrevue, un acteur montrait avec ses mains la manière dont il « descendait » dans le code, geste sur lequel nous l'avons questionné :

> *Q- C'est visuel, vous le voyez dans votre tête, "descendre" dans le code, de plus en plus?*
> Parce que la notion de descendre, c'est comme une notion de hiérarchie dans les objets, c'est une notion de descendre, on descend, on descend. Cette notion-là, elle existe dans la liaison entre les différents fichiers et dans les différents objets du code. Il y a toujours un parent, un enfant, ainsi de suite (sf04).

L'une des principales métaphores spatiales utilisées par les acteurs est celle de l'architecture. Comme on l'a vu en problématique, Lessig (2006) compare directement le code à une forme d'architecture qui limite ou favorise certains comportements. La métaphore de l'architecture se retrouve explicitement dans quelques-unes de nos entrevues. Un acteur affirme par

exemple que la période de 2007 a été consacrée à l'amélioration de l'architecture du code : « on a beaucoup travaillé sur le code, sur l'architecture du code dans symfony » (sf02). La métaphore de l'architecture est également sous-entendue dans l'utilisation des « *design patterns* », une des bonnes pratiques de développement auxquels font référence, de façon répétée, les acteurs de symfony.

D'une manière similaire, la notion de *couche* est également fortement mobilisée par les acteurs, qui décrivent la manière dont l'application constitue une superposition de différentes couches. L'un des acteurs décrit par exemple que le code source de symfony, en soit, ne fait rien, mais que ce sont plutôt les couches « supérieures » qui lui donnent vie :

> Symfony, ça ne fait rien. Par contre, il y a moyen de lui donner vie, en créant, en écrivant du code, un autre code source, qui va exploiter symfony. [...] C'est le code source qu'on va produire au-dessus, si on marche en termes de couches. Il y a symfony qui est en bas, au-dessus, il y a le code source applicatif que le développeur va produire. Et il va utiliser symfony comme base pour son programme (sf09).

D'une manière similaire, un acteur décrit l'architecture de SPIP en termes d'étages ou de couches :

> Il y a plusieurs étages dans le code de SPIP. Tu as un étage qui est le truc, ce qu'on appelle d'ailleurs le compilateur. [...] Tu as une deuxième couche autour de ça qui sont justement toutes les fonctions qui traitent directement avec ce code, ce compilateur. Ça c'est le *core*. C'est LE logiciel SPIP lui-même. Puis tu as un troisième niveau là-dedans, qui sont tous les points d'entrée qui sont aménagés dans ce logiciel qui permettent d'écrire des fonctions spécialisées qui ne font pas partie du logiciel SPIP lui-même, mais qui permettent de modifier son fonctionnement. [...] Et puis on pourrait presque dire un quatrième étage, qui sont liés à la nature, sa fonction, du logiciel. Qui est toute la méthode des squelettes, *templates*, gabarits, qui vont habiller les informations contenues dans la base de données, et qui est de calculer les pages Internet (spip07).

La métaphore du supermarché a également été mobilisée par cet acteur, mais cette fois, pour appréhender davantage ce qui nous semble être une spatialité « horizontale » du code source (au contraire de la notion de couche, qui met plutôt de l'avant une dimension « verticale »). Cette mobilisation de la métaphore du supermarché met de l'avant une certaine idée de classification des différentes parties du code source, où les fichiers et les noms de fonctions sont organisés de façon à permettre une plus grande efficacité, à la fois pour la machine et pour la personne qui participe à l'écriture du code :

> Dans un supermarché, tu as des présentoirs, des gondoles qu'on appelle ça, techniquement, dans lesquels tu as les produits présentés, par catégories, les lessives, les légumes, les fruits, le frais, les liquides, etc. Quand tu rentres dans le supermarché, selon ce que tu viens chercher, ce dont tu as besoin, tu vas avoir un certain parcours. Tu n'es pas obligé de passer partout. L'organisation du code, c'est pareil. [...] Si tu segmentes ces différentes fonctions, elles sont pas du tout rangées n'importe comment, elles ont un ordre, chaque fichier, il a 4-8-10 fonctions, qui sont cohérentes entre elles. Et donc, à partir de ce moment, ça permet de segmenter, d'alléger le fonctionnement de ton code, et c'est plus facile à réapprovisionner, pour reprendre la métaphore du supermarché. Le mec qui va réapprovisionner le rayon des liquides, il bouche le passage pour ce rayon, mais il permet à partout ailleurs que ça continue de circuler. Et donc, tu es pas obligé d'arrêter la vie du magasin quand tu veux aller faire un petit changement à cet endroit-là (spip07).

Cependant, selon un des acteurs avec lequel nous nous sommes entretenus, davantage que la métaphore de l'architecture, ce serait plutôt une métaphore de type « urbanistique » qui conviendrait davantage au projet de développement de logiciels de type collaboratif. Pour cet acteur, par ailleurs lui-même enseignant-chercheur en informatique :

> Ce qui est frappant en informatique, c'est que le mot architecture est très fréquent, on parle d'architecture de processeur, de logiciel, etc. Moi je dirais que le code informatique, dans le cadre d'un développement de logiciels collaboratif, il y a une dimension urbanistique. C'est une communauté qui a besoin de quelque chose qui fonctionne pour que la communauté et les visiteurs s'apportent des choses, puissent vivre ensemble. Et comme c'est du langage, la métaphore qui vient, c'est l'œuvre littéraire. Mais celle encore plus pertinente, c'est la métaphore urbanistique, c'est-à-dire qu'il y a une communauté qui construit une ville et cette ville n'est jamais finie. Quelqu'un a fait un bâtiment nouveau, qui s'inscrit dans un tissu urbain, et le bâtiment d'à côté il va être détruit plus tard, etc. Le développement de logiciels se rapproche beaucoup de ça. Il y a un certain nombre de gens qui prennent possession du code informatique qui est bourré des trucs, des bouts de bâtiment, chaque fichier est une espèce d'architecture qui a une existence un peu autonome et qui communique avec les autres. On peut dire que les signatures des fonctions, c'est comme les routes des villes, les rues qui vont permettre aux bâtiments de s'interpeller, de passer d'un bâtiment à l'autre, etc. Pour moi, c'est ça le code informatique. Quand on l'écrit tout seul, ça a un côté architectural, mais à plusieurs, c'est vraiment la métaphore urbanistique. C'est le vouloir vivre ensemble ! (spip03).

La métaphore urbanistique nous apparaît intéressante car elle permet d'aborder le code comme espace, mais également comme espace de « cohabitation » entre les différents codeurs. Pour cet acteur, le code, écrit à plusieurs, a un côté de « vivre ensemble » similaire à

une ville. Ces propos font à notre avis bien ressortir l'intérêt d'étudier le code source en sciences sociales. À la manière d'une ville, le code source constitue en quelque sorte l'habitat de la communauté qui vit dans le code source et le fait vivre. Le code source participe au lien social. Toutefois, cette dynamique de cohabitation, décrite par notre acteur, entraîne nécessairement le besoin de certaines règles pour assurer une certaine cohésion dans le code. Ainsi, il compare en ces termes les règles dans SPIP et les règles de construction des bâtiments dans une ville :

> Pour SPIP, c'est le même problème d'occupation des sols. Il faut trouver des règles de rédaction, observées par tous les acteurs du projet et qui vont donner une unité au code, même si c'est pas toujours la même chose, comme dans une ville bien conçue, où les bâtiments sont différents, avec une cohésion. À Paris, la limite des huit étages fait qu'on a une ligne d'horizon harmonieuse, même avec un bâtiment moderne à côté d'un médiéval. Cette ligne d'horizon donne un aspect reposant ! On a besoin de trouver quelque chose comme ça dans Spip, et on n'a pas trouvé (spip03).

Ces règles de rédaction feront notamment l'objet du chapitre suivant.

4.3 Les enjeux politiques de la définition du code source

La manière dont est défini le code source comporte certaines implications politiques, notamment dans la valorisation ou la dévalorisation de certaines activités en fonction de leur lien avec ce qui est considéré comme du code source. Nous avons par exemple noté dans le chapitre méthodologique la manière dont, lors du recrutement des participant-es à nos entrevues, on nous dirigeait systématiquement vers ceux (jamais celles) considérés comme les plus compétents en matière de code. De la même manière, certaines personnes que nous avons approchées, bien qu'étant parfaitement consentantes à participer à notre recherche et heureuses de pouvoir nous aider, ne voyaient pas du tout la pertinence de leur participation puisqu'elles considéraient ne rien connaître au code. Cette préoccupation quant aux conséquences politiques de la définition du code – et du code source – apparaît assez clairement dans les propos de cet acteur que nous avons cité plus tôt (spip01), qui souhaitait donner une définition très large du code source, de façon à pouvoir valoriser une plus grande diversité d'activités. Rappelons en effet que la préoccupation de cet acteur concerne le fait que certaines personnes au sein de SPIP déprécient leurs contributions dans le projet, parce qu'il ne s'agirait pas de contributions sous forme de code source (p. 140, section 4.2.1).

La manière dont les définitions sont liées à des dynamiques d'ordre plus politique est bien documenté dans la littérature. Nous avons par exemple déjà mentionné l'analyse que fait Serge Proulx (2007c) des métaphores de la société de l'information. L'auteur soutient que toute métaphore, ou terme employé par les acteurs, organise une certaine réalité et est également susceptible de devenir l'objet de controverses politiques[104]. Certaines auteures féministes ont également insisté sur le caractère politique des définitions. Pour Wajcman (2004) par exemple, le concept même de technologie est sujet aux changements historiques : à différentes époques, le concept de technologie désignait différentes choses. Ce n'est que dans le cadre de la formation de la profession d'ingénieur, depuis la fin du 19e siècle que la définition moderne de la technologie en est venue à être associée aux « machines masculines » industrielles plutôt qu'à l'artisanat féminin. Pour Wajcman, ce qui est défini comme technologie change avec le temps et, ce qui est plus important, les femmes ont joué un rôle important, et peuvent toujours jouer un rôle important dans les changements de la définition de la technologie : « Technological advances do open up new possibilities because some women are better placed to occupy the new spaces, and are less likely to regard machinery as a male domain » (Wajcman, 2004, p. 119).

C'est dans la perspective d'une prise en compte de la définition de ce qui constitue le code source (et le code en général) que nous élaborons cette dernière partie. Dans la première section, nous aborderons la catégorie du « codeur » en insistant sur les différentes hiérarchisations qui s'opèrent autour de cette catégorie. La deuxième section tentera pour sa part de rapporter ces différentes hiérarchisations autour des différentes formes et statuts de ce qui constitue le code source.

4.3.1 La catégorie du codeur

La catégorie du *codeur* revient souvent dans le discours des acteurs, autant dans nos entrevues que dans les échanges sur les listes de discussions. Cette catégorie est souvent mobilisée de façon neutre, pour faire référence simplement à quelqu'un qui fabrique ou produit du code. Cependant, à d'autres moments, cette catégorie marque le statut de certains acteurs.

---

104 Citant Bourdieu, Proulx écrit même que « Ce que l'on appelle des luttes de classes sont en fait des luttes de classement » (Bourdieu, 2000; cité par Proulx, 2007c, p. 113).

Par exemple, un acteur de SPIP que nous avons rencontré utilise ainsi la catégorie du codeur pour décrire certains acteurs de SPIP : « Par exemple, J et D[105], ils n'ont pas de problème avec le code en tant que tel, ce sont des codeurs [...], ils ont une approche technique, ils vont coder directement » (spip10). Nous avons cherché à en savoir plus sur la manière dont il définit cette catégorie, en lui demandant d'abord s'il se considère lui-même comme un codeur :

> *Q- Toi, tu te considères comme un codeur ? Tu mentionnais que J et D sont des codeurs ?*
> Hum. Je suis codeur, mais je ne suis pas un expert SPIP. Je suis pas un codeur à leur niveau. Effectivement, je code parce que c'est mon métier. Mais je n'ai pas une maîtrise suffisamment poussée de comment fonctionne SPIP au niveau du moteur du noyau, que je vais plutôt intervenir sur la périphérie, ou je n'interviendrai pas du tout là-dessus (spip10).

Cet extrait d'entrevue exprime une certaine hiérarchisation de la catégorie du codeur dans un contexte de spatialité du code source. Bien que codeur, cet acteur se distingue lui-même d'autres codeurs (les vrais ?) par le fait qu'il ne maîtrise pas suffisamment SPIP au niveau du *moteur,* du *noyau*, préférant plutôt intervenir en *périphérie*. Cette réponse ne nous satisfaisant pas, nous l'avons ensuite interrogé sur le statut de codeur de l'un des fondateurs du projet, encore aujourd'hui un acteur incontournable :

> *Q- Et G, tu considères que c'est un codeur ?*
> Non plus. C'est un bon codeur. Mais le code n'est pas la problématique qui l'intéresse. Le code est une solution pour répondre à une problématique, c'est pas une finalité en tant que tel (spip10).

Dans cette perspective, la caractéristique du codeur serait de prendre le code comme une finalité en soit. Nous lui avons demandé de confirmer notre analyse :

---

105 Afin de conserver l'anonymat des acteurs, ces initiales ne correspondent pas aux noms mentionnés.

> *Q- Tu définirais les codeurs comme ceux qui ont le code comme finalité ? Comment définirais-tu le terme codeur ?*
> Codeur, ouais c'est une bonne question, bon piège! Car même J dans ce cas-là ne serait pas un codeur non plus. Un codeur ce serait quoi ? [pense] C'est déjà avoir une vision globale technique, dans notre cas de SPIP, c'est-à-dire de comprendre comment fonctionne l'ensemble, et comment interagit l'ensemble, et ça, c'est pas donné à tout le monde [...]. Et après, c'est cette vision, d'avoir cette vision, de comprendre comment tout s'imbrique. Parce qu'après, on a surtout une vision périphérique, une vision spécifique. À ce moment-là, D a une vision plus globale que J parce que J est vraiment sur le compilateur. G, le code en tant que tel, c'est pas nécessairement [son truc], c'est plutôt comment le tout fonctionne et interagit avec les gens, etc. Je pense que les approches ne sont pas tout à fait les mêmes (spip10).

Cet extrait d'entrevue met de l'avant trois types d'acteurs. D'abord, D qui a une vision plus globale du fonctionnement de SPIP. J qui a une compréhension très poussée d'un élément crucial du mécanisme – le compilateur – sans toutefois avoir la même vision globale que D. Finalement, G qui s'intéresse plutôt à « comment le tout fonctionne, et interagit avec les gens », mais qui n'est cependant pas un codeur (selon notre interlocuteur)[106]. En terminant cette partie de la discussion, notre interlocuteur a proposé de distinguer les Codeurs, avec une majuscule, pour décrire ces gens qui portent « cette vision, de comprendre comment tout s'imbrique », et les distinguer des autres – les codeurs – qui font du code sans être nécessairement animés par cette vision[107].

Nous avons finalement interrogé notre interlocuteur concernant les relations entre l'autorité et cette catégorie du « codeur », et il apparaît que les liens sont complexes. Ainsi, pour cet acteur de SPIP et d'autres que nous avons rencontrés, c'est plutôt G, qui ne se qualifie pas comme un Codeur, mais qui s'intéresse à « comment le tout fonctionne, et interagit avec les gens » qui possède le plus d'autorité au sein du projet (bien que cette autorité ne soit pas explicitement formalisée). Si nous acceptons les propos des acteurs, dans le cas de SPIP, l'autorité ne revient pas tant aux Codeurs, qui mettent en relation les machines entre elles,

---

106 La figure du codeur décrite par cet acteur, qui aurait « cette vision, de comprendre comment tout s'imbrique » (spip10), semble rejoindre la figure du *mécanologue* chez Simondon, dont la préoccupation première ne concerne pas les usages, mais plutôt la relation des êtres techniques les uns par rapport aux autres (Simondon, 2001).

107 Cette perspective pourrait rejoindre les propos d'un autre acteur, de symfony cette fois (sf02), qui distingue pour sa part deux types de modification au code : les petites qui sont par exemple la correction de bugs ou les « mini-fonctionnalités » et les grandes évolutions qui impliquent des changements plus importants à l'architecture globale.

bien que ceux-ci occupent une place privilégiée, voire admirée dans le projet. Dans SPIP, l'autorité semble plutôt être détenue par celui qui réussit à mettre en relation les machines et les humains, et les humains entre eux.

La catégorie du codeur dans symfony

Au sein de symfony, il n'y a pas une aussi grande « mythification » de la catégorie du codeur, mais on peut tout de même constater une certaine hiérarchisation autour de cette catégorie. Ainsi, dans symfony, plusieurs acteurs font la différence entre les professionnels et les « bidouilleurs », cette dernière catégorie étant posée comme synonyme de « codeurs du dimanche » :

> J'ai envie de dire, la communauté des bidouilleurs, c'est pas péjoratif, ou des codeurs du dimanche. Dont c'est pas forcément le métier, mais qui utilisent PHP parce que c'est très simple à apprendre (sf02).

Pour cet acteur, il existerait une véritable séparation dans le monde PHP entre deux catégories de développeurs, soit les « bidouilleurs », ces « codeurs du dimanche » dont ce n'est pas forcément le métier, et les professionnels qui utilisent PHP en entreprise et qui, selon cet acteur « souhaitent utiliser PHP comme ils utilisent .net ou java » (sf02). Cette séparation serait, toujours selon cet acteur, davantage marquée depuis l'arrivée de la version 5 de PHP, qui viserait un public cible plus professionnel, « de gens qui ont un bagage technique assez important » (sf02)[108]. Pour cet acteur, la sortie de PHP5 se serait accompagnée du développement de plusieurs outils beaucoup plus professionnels, dans lequel s'inscrit

---

108 Dans le cadre d'une conférence que nous avons réalisée au début de notre doctorat (Couture, 2008), nous avions analysé le cas du mouvement *GoToPHP5* qui semble, en regard des propos de cet acteur, également marquer une certaine rupture entre ces deux catégories de codeurs PHP. Le mouvement GoToPHP5 réunissait différents projets de logiciels libres et d'hébergeurs en vue de faire adopter la version 5.0 de PHP comme un standard, et d'abandonner le développement de PHP 4. En effet, la version 5 de PHP incluait différentes fonctionnalités appropriées pour les applications de web 2.0, mais celle-ci était encore relativement instable, du fait notamment que l'équipe de PHP devait maintenir la version de PHP 4 étant donné le nombre important de projets et d'hébergeurs qui utilisaient encore cette version. L'objectif du mouvement GoToPHP5 était donc de réunir le plus grand nombre possible d'hébergeurs et de projets de logiciels libres qui abandonneraient le support à PHP4, pour le 5 février 2008 (la date a été choisie car elle représentait le 5e jour du 2e mois de l'année, ce qui symbolisait par conséquent la version de PHP 5.2, vers laquelle la migration était souhaitée). Parmi les projets de logiciels libres qui ont participé à ce mouvement, on peut compter *Drupal*, *Moodle*, *PHPMyAdmin*, et également *Doctrine* et *Propel*, qui sont désormais étroitement liés à symfony. *SPIP* n'a cependant pas adhéré à ce mouvement.

symfony, mais également d'autres technologies de type Framework, tels que le *Zend Framework* ou *EasyComponents*. Selon cet acteur, symfony est un projet qui devrait être destiné à ce type de clientèle :

> Ce qui fait que la cible de symfony ou du Zend Framework, typiquement ces deux-là, ou *EasyComponents*, est une cible très restreinte par rapport à la communauté globale de PHP. C'est une cible plutôt professionnelle, plutôt de gens dont c'est le métier. Donc plutôt de gens qui ont un bagage technique assez important (sf02).

D'une certaine manière, nous pourrions dire que cette distinction marque aussi la distinction entre SPIP et symfony. C'est ainsi qu'un acteur dit que SPIP est codé « à l'ancienne » (sf08). Au niveau temporel, comme nous l'avons écrit dans le chapitre méthodologique, SPIP est beaucoup plus âgé que symfony. SPIP est en effet né en 2001, alors que symfony est pour sa part né en 2007.

4.3.2 Politique et pragmatique du code source

Cependant, notre intérêt ici ne porte pas tant sur la catégorie du codeur, mais plutôt sur le code source, appréhendé en tant que catégorie, mais également en tant qu'artefact. C'est donc plutôt vers une pragmatique du code source que nous devons nous tourner. Plusieurs des extraits d'entrevues permettent de mettre en relation cette manière dont une partie donnée du code est associée à un enjeu politique. Ainsi, pour citer de nouveau un acteur que avons abordé plus tôt, l'une des choses qui distinguerait les « vrais codeurs » est le fait que ce sont des codeurs du *core* de SPIP[109] :

> Mais au sein du *core*, on fait quand même bien la distinction entre les vrais codeurs, qui se comptent sur les doigts d'une main, enfin, les vrais codeur du code, du *core* de SPIP, et les autres (spip08).

---

109 Nous n'avons pas, à ce stade-ci, une théorie concluante expliquant les raisons de ce qui semble être une mythification du code plus grande au sein de SPIP que de symfony. Deux hypothèses pourraient cependant être envisagées. La première concerne l'hétérogénéité des compétences, plus grande dans SPIP que dans symfony. En effet, alors que la plupart des acteurs que nous avons rencontrés dans symfony possèdent une formation de premier cycle en informatique, la situation est très variable dans SPIP où certains acteurs enseignent l'informatique théorique à l'Université, tandis que d'autres n'avaient qu'une très faible expérience dans l'usage des ordinateurs avant de joindre la communauté SPIP. Une autre hypothèse expliquant cette disparité concerne la grande réflexivité de certains acteurs de SPIP que nous avons rencontrés, qui avaient déjà réfléchi à toutes ces questions.

Ainsi, pour cette actrice, ceux qui ne contribuent pas au code du *core*, sont en quelque sorte « illégitimes » (c'est le mot qu'elle emploie) par rapport au code :

> Illégitime, au sens de codeur PHP. [...] En fait, illégitime, ça veut rien dire, mais illégitime, par rapport au code. Je sais très bien, et on est certain à savoir très bien, qu'on n'est pas capables de coder une ligne dans le *core* du logiciel (spip08).

Ce qui est intéressant dans cet extrait d'entrevue, c'est que ce sentiment d'illégitimité, de ne pas être un « vrai codeur », n'est pas tant lié à la compétence de l'acteur, comme c'était le cas plus tôt dans notre description du Codeur, avec une majuscule. Le fait ou non d'être un « vrai codeur » est lié, du moins dans cet extrait d'entrevue, à la position, ou au statut du code auquel on est « capable » ou non de contribuer, dans ce cas-ci, le « core ». Nous verrons au chapitre 6 que cette capacité n'est pas seulement liée à la compétence, mais également au fait d'être autorisé ou non à modifier telle ou telle partie du code source. Plus tôt dans l'entrevue, cette actrice, qui disait ne rien connaître au code, nous expliquait ainsi la distinction qu'elle faisait entre le « code SPIP », écrit dans un langage simplifié qui n'est pas du langage (selon elle), et le code écrit dans le langage PHP :

> Ouais, mais le code SPIP, mais qui est complètement différent du code du langage objet, ou du langage PHP [...] Je ne sais pas du tout comment on programme. [...] Moi je suis autodidacte, le langage SPIP, les CSS et tout ça, j'y arrive, mais parce que c'est pas du langage, c'est du langage simplifié en fait (spip08).

Dit en d'autres termes, pour cette actrice, non seulement le « code SPIP » n'est pas de même type que le code PHP, mais à certains égards, il n'est même pas du code. Cette manière dont la définition même de ce qui relève ou non du code s'articule à certains positionnements sociaux est d'ailleurs reconnue comme problématique par d'autres acteurs de SPIP :

> *Q- Est-ce que tu penses que ça peut avoir une implication dans le projet, la manière dont les différentes personnes définissent le « code source » ?*
> Oui, sûrement. En tout cas, il y a beaucoup de gens qui minimisent leur apport, ou leur participation, en pensant qu'ils ne font pas du code, ou je sais pas quoi. C'est tout le temps comme ça les discussions (spip01).

Cet acteur est celui que nous avons présenté plus tôt et qui insistait sur la nécessité de donner une définition très large au code source. Celui-ci cherche en effet à élargir la définition du code source, de façon à inclure un grand nombre d'activités, telles que faire des maquettes fonctionnelles ou bien créer de la documentation. Plus tard dans l'entrevue, notre

interlocuteur précise certaines distinctions, en parlant cette fois du « code informatique, au sens de langage de programmation », qui aurait selon lui une caractéristique « magique », ce qui n'est pas le cas de la maquette graphique, par exemple. Nous reproduisons ici complètement cette partie de l'entrevue, particulièrement intéressante en soulignant les passages les plus importants :

> - Plus le fait effectivement que **le code informatique, au sens de langage de programmation**, c'est... c'est plus abstrait, c'est plus... [silence] Ça a un lien avec la magie. C'est pas évident de savoir.
>
> *Q- Ça a un lien avec la magie ?*
> R- Ben oui. C'est des incantations magiques.
>
> *Q- Le code informatique ?*
> - Ben oui.
>
> *Q- Pourquoi ? Ça te semble évident ?*
> - Ben, tu prononces des mots et ça produit un effet. C'est pas ça la magie ? Il faut savoir quels mots, il faut savoir quels effets. Quelle suite de mots pour obtenir tel effet, etc. C'est ça, dans *Harry Potter*, c'est exactement ça. Ils apprennent des mots, utiliser le bon mot au bon moment pour provoquer l'effet sur le milieu qui t'environne.
>
> *Q- Et ça, est-ce que ce serait différent de faire une maquette, ou faire de la documentation ?*
> - Bien oui. Tu as au moins cette notion-là puisque tu...
>
> *Q- Cet aspect magique ?*
> - Oui. C'est une magie en même temps. Quand tu maîtrises bien le langage, tu vois bien que ce n'est pas de la magie. Mais quand tu ne maîtrises pas bien, tu vois les mots, tu vois à peu près seulement... La manière dont ça s'enchâsse, les conséquences de chaque mot, tu ne maîtrise pas complètement. Donc, si ça marche, c'est un peu par magie. Donc les gens ont tendance à minimiser leurs trucs surtout que tu peux passer une journée entière à faire trois lignes, parce que tu n'arrives pas à les mettre bien. Je pense que ça, c'est... **Alors qu'il n'y a pas la même approche par rapport au graphisme.** Quand tu ne sais pas faire du graphisme, tu peux quand même faire une icône. Elle ne sera pas belle, elle ne sera pas fonctionnelle, mais elle ne sera pas en erreur de syntaxe. Il n'y a pas la même sanction quand c'est raté (spip01).

Interprété dans notre cadre conceptuel, nous pourrions dire ici qu'à la différence du graphisme, même agissant à titre de spécification, le « code informatique, pris au sens de langage », a comme caractéristique particulière d'être en soi performatif. Les propos de notre interlocuteur « tu prononces des mots, et ça produit un effet » (sf01) résonnent en effet de

façon frappante avec le concept de performativité d'Austin – quand dire c'est faire. L'une des questions qui nous préoccupe dans cette thèse, et que nous aborderons au chapitre 6, est de comprendre plus précisément les conditions de cette performativité.

4.4 Conclusion partielle : le code source, un artefact aux frontières ambiguës

Notre objectif dans cette thèse n'est pas d'en arriver à une définition formelle du code source ou encore, d'insister trop longtemps sur toutes les nuances conceptuelles possibles dans la définition de code source. Notre objectif est plutôt d'analyser *l'artefact* qui est compris de façon commune par les acteurs, comme étant le code source. Cette façon de faire rejoint l'approche pragmatiste adoptée par Bowker et Star dans leur étude des classifications, consistant à concentrer l'attention sur l'artefact lui-même et sur le travail de construction et de maintenance qui lui est associé, plutôt que de tenter de purifier le concept de code source :

> For the purposes of this book, we take a broad enough definition so that anything consistently called a classifcation system and treated as such can be included in the term. This is a classic Pragmatist turn – things perceived as real are real in their consequences. [...] With a broad, Pragmatic definition we can look at the work that is involved in building and maintaining a family of entities that people call classification systems rather than attempt the Herculean, Sisyphian task of purifying the (un)stable systems in place (Bowker et Star, 2000a, p. 13).

Mentionnons d'abord que l'usage même du terme de code source est problématique pour les acteurs. En effet, comme nous l'avons noté à la fin du chapitre précédent et dans le cours de celui-ci, les termes de « code » et de « code source » semblent souvent utilisés de façon synonyme par les acteurs. Un acteur de SPIP nous mentionne même que la notion de « code source » n'a pas de sens dans le langage PHP, puisqu'il ne s'agit pas d'un langage compilé. Cette position est cependant isolée et, comme nous l'avons montré, le terme de « code source » est effectivement mobilisé par les acteurs, notamment sur certaines pages web des projets (en particulier SPIP). De plus, le fait que tous les acteurs que nous avons rencontrés en entrevue ont quelque chose à nous dire sur le « code source », démontre au moins que ce terme a pour eux une certaine signification.

Ceci étant dit, les frontières de la notion de « code source » restent ambiguës. C'est d'abord le cas du caractère écrit du code source. Alors que plusieurs acteurs considèrent que le code source est un écrit, un « texte », certains apportent quelques nuances, en mentionnant par

exemple que le code source peut parfois prendre la forme d'un « schéma-bloc » graphique. D'autres donnent une définition très générale du code source. Pour l'un des acteurs, est code source ce qui définit formellement le fonctionnement d'un logiciel, voire d'un système informatique, comme Internet. Dans cette perspective, les *Request for Comments* constitueraient en quelque sorte le code source de l'Internet, même si celui-ci n'est pas directement exécuté par la machine. Cet acteur établit toutefois plus tard une distinction entre cette définition large du code source et celle plus précise du « code informatique, pris au sens de langage ».

Le statut de la documentation en regard du code source est également complexe. Pour certains acteurs, le code source ne peut exister sans sa documentation, dans ce sens qu'un morceau de code source non documenté ne sera simplement pas utilisé. Plus concrètement, plusieurs acteurs ont abordé le statut des « commentaires ». Font-ils partie du code source puisqu'ils sont inscrits dans les mêmes fichiers que les instructions exécutables? Qu'en est-il de ces lignes de code qui agissent comme commentaire dans un contexte, mais servent de « code source » dans d'autres contexte ? Nous avons finalement constaté que la définition même du code source revêt un caractère politique pour certains acteurs, car le fait de définir tel ou tel type d'activité comme du codage, et par conséquent, tel ou tel type d'artefact comme du code, a des conséquences sur la valorisation de l'activité.

Si les frontières définitionnelles du code source sont ambiguës, ses frontières « artefactuelles » le sont également. Dans la première partie du chapitre, nous avons cherché à analyser les différentes formes et statuts du code source. Nos descriptions font ressortir qu'autour du code source du cœur – ou du *core* – le code source « que je peux télécharger », on retrouve une pluralité d'espaces regroupant du code source ayant différentes formes et statuts. Dans chacun des projets, on remarque par exemple une quantité importante de plugins, dont le « code source » est distinct de celui du *core*, mais qui peut cependant s'articuler à celui-ci. Cette analyse fait ressortir le caractère particulièrement dispersé du code source, qui prend une diversité de formes et de statuts dans les projets étudiés. Dès lors, la question est de savoir comment donner une cohérence à cette dispersion, comment faire circuler ces « bouts de code source » et éventuellement les faire converger, de façon à obtenir un code source effectivement stable, propre à être éventuellement exécuté et utilisé.

L'analyse de cette compréhension commune de ce que constitue le code source des projets étudiés permet d'en faire ressortir quelques traits. Ceux-ci nous permettent de circonscrire notre objet d'étude et d'orienter notre analyse pour la suite de la thèse :

- Le code source est fondamentalement relationnel. Dans les projets étudiés, le code source n'existe jamais pour lui-même; il est toujours le code source de quelque chose d'autre, d'un artefact computationnel telle une application ou un logiciel.

- De façon générale, nous postulons que le code source informatique est généralement appréhendé « au sens de langage de programmation ». Selon cette définition, le code source informatique est principalement un « code qu'on écrit ». C'est du texte, découpé en plusieurs fichiers et en plusieurs morceaux, selon des normes que nous explorerons dans le prochain chapitre.- Pris au sens commun de « langage de programmation », le code source a la double propriété d'être fabriqué par des humains et d'être exécuté par un ou des ordinateurs. Pour reprendre certaines métaphores des acteurs, c'est le code « qu'on va pétrir » et qui aura un effet « magique » : il fait agir un ordinateur. Ce dernier aspect renvoie justement à l'aspect performatif, voire « magique », du code source informatique : ce sont des mots qui produisent des effets. La question, d'un point de vue performatif, est donc de savoir quels mots, agencés de quelles manières, et dans quel environnement, produisent quels effets. C'est ce que nous continuerons à explorer dans la suite de la thèse.

CHAPITRE V

LA « BEAUTÉ DU CODE ». NORMES D'ÉCRITURE ET QUALITÉS DU CODE SOURCE

> Je me mets à la place du développeur et je me dis, ah, c'est du beau code, quand on le voit. Mais quand on commence à l'utiliser finalement c'est pas très pratique ! Ils auraient pu le faire un peu moins beau, mais plus facile à utiliser ! C'est des contradictions qui deviennent fondamentales quand elles sont érigées en principe (sf03).

Le point de départ du chapitre concerne certaines catégories utilisées par les acteurs pour apprécier la qualité « interne » du code source. Ce choix méthodologique permet de saisir le code source dans son mode d'existence propre (Simondon, 2001), dans ce qui est en quelque sorte « effacé » par la transformation du code source en logiciel exécutable. Après quelques précisions sur l'origine de notre questionnement sur la beauté du code en première partie du chapitre, la deuxième partie s'intéresse aux catégories telles que la beauté, l'élégance, la propreté ou la lisibilité, mobilisées par les acteurs pour juger de cette qualité du code source. La troisième partie aborde certaines normes, règles et « bonnes pratiques » qui participent, selon les acteurs, à la qualité du code source. La quatrième partie analyse deux controverses qui mettent en relation différentes catégories de la qualité et une certaine configuration de l'usager (Woolgar, 1991). En conclusion, nous nous appuyons sur les propos d'un acteur pour considérer le code source à la manière d'une interface à travers laquelle les acteurs interagissent avec le logiciel, ou certaines parties du logiciel.

5.1 La « beauté du code » ou la dimension expressive du code source

Ce chapitre a pour origine un questionnement sur la « beauté du code », qui nous a particulièrement préoccupés en début de thèse au point où, comme nous l'avons mentionné en avant-propos, nous avons consacré un travail de session sur ce thème spécifique. Cette préoccupation prenait racine d'une part, dans le constat présenté en problématique, de l'importance donnée par les informaticien-es, hackers et autres programmeurs, à une certaine

dimension expressive, voire même esthétique, de leur activité. L'ouvrage *Beautiful Code* (Oram et Wilson, 2007), cité en problématique, est assez significatif de cette dynamique, comme le sont les propos de Donald Knuth, sur la programmation comme forme d'art. Rappelons que pour Knuth, écrire un programme peut constituer une expérience esthétique comme composer un poème ou de la musique (Knuth, 1968; Knuth, 1974; Krysia et Grzesiek, 2008). Notre intérêt pour cette catégorie de la beauté permet également de rejoindre les réflexions de Simondon sur les modes d'existence des objets techniques. Comme il a également été mentionné au chapitre 2, Simondon consacre de très belles pages aux rapports entre technique et esthétique, en abordant explicitement la question de la « beauté des objets techniques » :

> C'est pourquoi la découverte de la beauté des objets techniques ne peut pas être laissée à la seule perception : il faut que la fonction de l'objet soit comprise et pensée; autrement dit, il faut une éducation technique pour que la beauté des objets techniques puisse apparaître comme insertion des schèmes techniques dans un univers, aux points-clefs de cet univers (Simondon, 2001, p. 186).

Rappelons que le projet de cet auteur consiste à susciter une prise de conscience du sens des objets techniques qui permettrait de les intégrer dans la culture. Pour Simondon, la culture refoule les objets techniques dans le monde de ce qui ne possède pas de signification mais seulement un usage, une fonction utile. Plutôt que de poser le regard sur les seules dimensions utilitaires ou instrumentales de l'objet technique, Simondon cherche plutôt à faire ressortir cette part de l'humain qui réside dans la machine (Simondon, 2001, p. 12). Dans ce sens, il nous semblait qu'appréhender la beauté des objets technique – ou le discours sur cette beauté – nous permettait de saisir la dimension non-instrumentale du code source. Comme indiqué en avant-propos, cette place accordée au thème de la « beauté du code » a ensuite été progressivement réduite dans notre étude. D'une part, il nous semblait nécessaire d'appréhender de façon plus large notre objet de recherche, le code source, en prenant en compte d'autres dimensions que la seule catégorie de la « beauté ». D'autre part, nous nous sommes rapidement aperçu que cette catégorie de la « beauté du code » était problématique parmi les acteurs des projets étudiés. Déjà, lors d'une entrevue préliminaire, un acteur qui n'était lié ni à SPIP ni à symfony, nous avons appris que le langage PHP, que nous avions choisi d'étudier, était considéré comme un langage « laid ». De la même manière, la catégorie de la « beauté » ne semble pas être fortement mobilisée parmi les acteurs de SPIP et symfony.

Ainsi, d'un point de vue méthodologique, il est par exemple notable de mentionner que c'est nous-mêmes qui avons introduit la catégorie de la beauté dans le cadre de nos questions au lieu qu'elle n'ait émergé d'elle-même de la part des acteurs. Cette catégorie semble donc en quelque sorte « exogène » au vocabulaire commun des acteurs et que nous attarderons plus loin à d'autres catégories plus « indigènes », telles que l'élégance, la propreté ou la lisibilité. L'extrait suivant montre bien comment la catégorie de la beauté interpelle certains acteurs :

> *Un autre qualificatif que tu n'as pas mentionné, c'est un code qui est « beau » ? Tu crois que ça s'applique au code ?*
> C'est marrant que tu dises ça [...], c'est vrai que c'est un truc que j'ai ressenti ! C'est un beau programme, de la même manière qu'on peut parler d'une belle démonstration en mathématique ! C'est significatif, j'avais jamais parlé du mot « beau » du code de SPIP, c'est justement ce qui me gêne et que j'aimerais arriver à faire, à le rendre beau ! Actuellement il ne l'est pas du tout ! (spip03).

Cet extrait d'entrevue montre bien la pertinence, au moins sur le plan heuristique, d'aborder la catégorie de la beauté auprès des acteurs. L'exploration de la catégorie de la beauté permet, pour reprendre les mots de Simondon (2001) - et plus récemment, de Latour (2010) - d'appréhender le code source dans son mode d'existence propre, et non pas seulement comme un moyen qui serait subordonné à une fin (la construction d'un logiciel exécutable). Notre approche rejoint aussi les propos de Fuller, qui cite Knuth, et pour qui l'élégance (une catégorie très similaire à celle de la beauté) permet d'appréhender la programmation comme une activité pour elle-même : « Knuth's criteria for elegance are immensely powerful when evaluating programming as an activity in and of itself » (Fuller, 2008, p. 90). Analyser la manière dont les acteurs définissent ou mobilisent des catégories telle la beauté, permet en quelque sorte d'ouvrir la boîte noire du logiciel et d'approfondir notre description de ce qui constitue le code source dans les projets étudiés. En outre, comme nous le verrons au cours de ce chapitre, ce questionnement initial sur la « beauté » nous a permis d'aborder d'autres

catégories reliées, telles que l'*élégance*, la *propreté,* la *lisibilité*, voire l'*odeur du code*, catégories que nous pourrions qualifier de qualités « internes » au code source[110].

5.2 Le vocabulaire de la qualité

Cette partie du chapitre aborde la manière dont les acteurs mobilisent ces différentes catégories de la qualité, que nous avons introduites précédemment. Nous présentons tout d'abord les catégories de la beauté et de l'élégance. Est ensuite abordée la catégorie de la lisibilité, une catégorie importante pour les acteurs, et que nous avons progressivement prise en compte durant notre étude. Nous terminons en exposant d'autres termes mobilisés par les acteurs pour décrire la qualité du code source, tels que la propreté et l'odeur du code source. À la fin de cette seconde partie du chapitre, nous présentons un tableau synoptique de ce vocabulaire de la qualité (tableau 5.1).

5.2.1 Beauté et élégance du code source

À quoi renvoie au juste la « beauté du code » chez les acteurs rencontrés en entrevue ? Comme mentionné plus tôt, dans la plupart des entrevues réalisées, c'est nous-mêmes qui avons introduit cette catégorie. Cette situation nous amène à poser l'hypothèse que la catégorie de la beauté n'a pas une grande importance dans les appréciations que les acteurs font de la qualité du code source.

Parmi ceux qui considèrent pertinent de parler d'une beauté du code, les idées de pureté ou de simplicité ressortent. L'un des acteurs de SPIP mentionne ainsi que la beauté, « c'est un côté simple, épuration » (spip04), ce serait une « beauté géométrique » où « chaque fichier a sa place, et des fichiers courts aussi, ça participe à la beauté » (spip04). Un autre acteur de SPIP donne l'exemple du langage de programmation Jquery, qu'il considère comme beau, parce qu'il serait « très expressif, plutôt en un minimum de mots » (spip01). Un acteur compare

110 La « qualité interne » est une notion que nous forgeons ici de façon ad hoc pour regrouper les différentes catégories décrites dans ce chapitre, telles que la beauté, l'élégance, la lisibilité, la propreté ou la clarté, qui nous semblent renvoyer à des qualités proprement « internes » au code source (au contraire par exemple de l'*efficacité,* une qualité davantage liée au mode d'existence « exécutable » du logiciel). Bien que la notion de qualité ici mobilisée résonne avec certains propos des acteurs, elle ne renvoie pas à un cadre théorique précis. Mentionnons toutefois l'étude de Pène (2005) sur les « agencements langagiers de la qualité » qui s'attarde dans une perspective de linguistique de terrain, à la mise en place des systèmes qualité dans l'industrie et dans les services.

aussi la beauté du code à « une belle démonstration en mathématique » (spip03). Pour cet acteur, l'idéal de la beauté en informatique serait le langage LISP qui s'exprimerait par une « beauté mathématique pure » (spip03). Pour lui, la beauté renvoie donc également à une sorte de pureté, voire à une certaine unité, ou à une harmonie.

Les idées de pureté et de simplicité sont également présentes chez les acteurs de symfony que nous avons rencontrés. Un des acteurs nous a explicitement mentionné « qu'une des conditions de la beauté, c'est la simplicité » (sf06) et compare la beauté du code à celle des arts martiaux japonais :

> On va épurer au maximum, dans les arts martiaux, on va essayer de faire le moins de mouvements possible. [...] Pour moi un beau code, c'est quelque chose qui va suivre cette philosophie : la recherche de l'efficacité maximale avec la plus grande simplicité possible (sf06).

Certains acteurs distinguent la beauté du code d'autres qualités que nous aborderons en détail plus loin. Ainsi, un acteur de SPIP soutient que la beauté relèverait plus de l'émotionnel, du subjectif, au contraire de la clarté et de la lisibilité (catégories que nous aborderons plus loin) qui relèveraient quant à elles davantage de l'intellectuel : « il y a clarté, il y a lisibilité, mais ça, c'est plus de l'intellectuel. Beauté, c'est plus émotionnel, c'est plus subjectif » (spip04)[111]. Un acteur soutient aussi que « quand c'est beau, c'est que c'est pas seulement propre, ou bien fait, mais il y a quelque chose d'un peu inattendu, et d'un peu astucieux » (spip01). Pour une actrice de symfony, la question de la beauté renvoie davantage au terme anglais de *cleverness* : « Wow! C'est beau! C'est vraiment malin » (sf04).

La catégorie de la beauté est cependant très problématique pour plusieurs acteurs, en particulier lorsqu'elle s'assimile à la pureté. Ainsi, cet acteur qui idéalise la « beauté mathématique pure » nous explique qu'il est difficile de faire du beau code avec le langage de squelettes SPIP, tout comme avec le langage PHP, car il s'agit de langages « métissés ». Cet acteur exprime cependant lui-même un certain malaise par rapport à ce terme :

---

111 Notons que cet acteur affirme d'ailleurs ne pas être en mesure, lui-même, de percevoir cette beauté, bien qu'il reconnaisse que des gens plus expérimentés puisse la percevoir.

> Maintenant, est-ce que c'est possible de faire du beau code... c'est pas très évident, dans SPIP, car ses squelettes, comme PHP, ça va paraître raciste ce que je vais dire, font partie des langages métissés. [...] On appelle ça métissé, c'est l'adjectif standard, je l'aime pas beaucoup mais... Quand on voit un métis, il a une peau d'une couleur intermédiaire entre celle de son père et de sa mère. Mais pour le langage métis, c'est plus comme si quelqu'un avait une partie de son corps avec la peau de sa mère et une autre partie avec celle de son père, comme un léopard. Cela dit, un léopard, c'est très beau ! (spip03)

Reprenant la métaphore urbanistique que nous avons mentionnée dans le chapitre précédent, cet acteur compare un code source qui serait beau à la ville de Paris, tandis qu'un code source qui serait laid à une ville du tiers-monde :

> Si il y a 40 000 personnes qui participent à la construction d'une ville ou du code de SPIP, alors c'est quoi la beauté d'une ville ? Je suis pas en train de dire, à la corbusienne, que les seules belles villes, c'est celles planifiées par une seule personne, toute l'architecture... Mais le plan d'occupation des sols, il y en a des réussis et d'autres qui ne le sont pas ! Soit ils sont trop contraints, à la Le Corbusier, soit trop laxistes ou inexistants comme les villes du tiers-monde qui sont horribles ! (spip03)

Notre objectif n'est pas ici de critiquer la conception de la beauté portée par cet acteur (qui est par ailleurs très réflexif dans ses propos). Il ne s'agit pas non plus de faire ressortir les représentations de la beauté « cachées » dans le discours. Simplement, nous voulons ici montrer que la catégorie de la beauté renvoie à différentes configurations du code source qui peuvent elles-mêmes être contestées, comme nous le verrons à la section 5.4. Un autre aspect à noter est que l'appréciation d'un même morceau de code semble être variable d'une personne à l'autre. Concernant le code source de SPIP, deux acteurs émettent par exemple des appréciations contradictoires :

> Je n'avais jamais parlé du mot « beau » du code de SPIP (spip03).

> *Q- Est-ce que tu penses qu'un code peut être beau ?*
> À mon avis, il tend à l'être. Au niveau de SPIP, il l'est (spip02)[112].

112 Concernant la beauté, en particulier dans le cadre de SPIP, il nous semble que les personnes les plus engagées dans le projet soient moins portées à considérer le code source comme étant « beau ». Mentionnons cependant les propos de cet acteur qui tendraient à infirmer notre hypothèse : « Je pense qu'une personne qui est plus investie, qui en fait un travail quasi quotidien, on va dire, peut facilement la percevoir [la beauté du code] » (spip04). Ces propos pourraient cependant renforcer notre hypothèse : les personnes les plus investies seraient en effet celles les plus à même de « percevoir » la beauté, mais a contrario, également la « non-beauté » d'un code source.

D'autres acteurs remettent également en question l'insistance de certains programmeurs sur la beauté. Une actrice de SPIP mentionne que plusieurs programmeurs qui insistent sur la beauté, ne sont pas de très bons collègues :

> J'ai rencontré des programmeurs qui soit étaient en admiration devant le code, soit pensaient que leur code était beau, et qui étaient du coup pas des bons collègues, parce que, humainement, ils étaient trop dans cette beauté, cette élégance (sf05).

Un autre élément à mentionner, et sur lequel nous reviendrons dans la dernière partie de ce chapitre, concerne le fait de mettre ou non l'accent sur la beauté, surtout si celui-ci se fait au détriment d'autres dimensions telles que l'ergonomie, ou l' « utilisabilité », voire même les relations interpersonnelles. Dans nos entrevues, c'est parmi les acteurs de symfony que cette préoccupation est la plus explicite :

> Je me mets à la place du développeur et je me dis, ah, c'est du beau code, quand on le voit. Mais quand on commence à l'utiliser finalement c'est pas très pratique! Ils auraient pu le faire un peu moins beau, mais plus facile à utiliser ! C'est des contradictions qui deviennent fondamentales quand elles sont érigées en principe (sf03).

À propos de la beauté, mentionnons finalement que certains acteurs ne considèrent pas cette catégorie pertinente pour appréhender le code. Un acteur de SPIP nous indique par exemple que « Pour moi, la beauté, c'est du vocabulaire de l'art. Et le code, ce n'est absolument pas de l'art » (spip07). Celui-ci préfère plutôt parler de l'élégance, une catégorie également mobilisée par les acteurs :

> *Chercheur : Il y en a qui parlent d'un code qui serait beau...*
> Élégant, le vrai mot, c'est élégant (sf07).

D'autres acteurs renvoient de façon plus ou moins explicite à la question du découpage entre les différentes composantes du code pour parler de l'élégance. Ainsi, l'un des acteurs de SPIP retrouve l'élégance du code source de SPIP dans son découpage entre différentes composantes, en utilisant des termes tels qu'« aménagé » et « à l'intérieur » qui rappellent encore une fois, une certaine représentation spatiale du code source :

> Ce découpage, la façon dont c'est écrit. La façon dont c'est aménagé. La compréhension qui est donnée à la fois de ce découpage, la compréhension de ce code, de cette lecture qui permet d'aller de plus en plus loin à l'intérieur, même si ce n'est pas toi qui a écrit le compilateur, petit à petit, tu peux aller de plus en plus loin dans la compréhension de comment ça fonctionne. Et tu peux toi-même t'immiscer là-dedans à des niveaux très différents, ça, je trouve ça élégant (spip07).

Un participant à nos entrevues (sf10) définit l'élégance de façon plus concrète en distinguant trois niveaux à cette élégance qu'il nous semble intéressant de présenter ici. Un premier niveau d'élégance concerne le style et la présentation et se rapproche du critère de lisibilité, sur lequel nous reviendrons dans la prochaine sous-section : « Du code élégant, c'est déjà bien présenté, c'est-à-dire que les commentaires sont en anglais, les variables sont en anglais, les variables sont intelligibles, l'indentation est respectée » (sf10). Le respect d'une « charte de nommage » ferait également partie de ce premier niveau d'élégance. Un deuxième niveau d'élégance serait pour cet acteur une « élégance de conception ». La simplicité relèverait par exemple de ce niveau d'élégance, ou encore, le fait de ne pas répéter plusieurs fois un même morceau de code. Il s'agirait également de respecter certains principes de conception, en particulier ceux préconisés au sein du projet[113]. Finalement, un troisième niveau d'élégance aurait trait à l'organisation du code. Il s'agit de « cloisonner » le code source en différentes parties en évitant les dépendances entre celles-ci : « Qu'il n'y ait pas trop de dépendances, sinon, quand je vais bouger la méthode d'à côté, elle va se casser la gueule aussi[114] » (sf10). Bien que cette description fasse référence à bon nombre de termes informatiques, tels que les variables, méthodes ou chartes de nommage, que nous n'avons pas encore abordés, il est assez aisé de constater dans cette description le caractère éminemment collectif de cette élégance. Le fait que les variables et les commentaires soient en anglais, et que le code doive respecter une « charte de nommage » – imposée par les dirigeants du projet – ou encore que certains principes de conception doivent être respectés, montrent bien l'importance de l'insertion correcte d'un bout de code source dans une chaîne d'écriture plus large. Nous

---

113 Cet acteur faisait référence ici principalement aux motifs de conception, appelés aussi *design patterns*, sur lesquels nous reviendrons à la prochaine section consacrée aux normes d'écriture du code source.

114 Une *méthode*, en programmation orientée-objet, est une portion de code source encapsulée de manière à répondre à un objectif fonctionnel précis. Il correspond à la *fonction*, en programmation procédurale.

verrons que cette insertion réussie d'un bout code dans cette chaîne d'écriture relève non seulement d'une élégance abstraite, mais constitue également en bonne partie les conditions de la performativité de la portion de code source en question.

5.2.2 La lisibilité, une catégorie émergente et significative

Contrairement à la beauté, qui était une catégorie initiale dans notre étude, celle de la lisibilité a émergé durant l'enquête. C'est d'abord la page web du projet symfony, qui mentionnait que ce projet se caractérisait par un code lisible et une conception propre (« *clean design and readable code*[115] »), qui a retenu notre attention pour analyser la catégorie de la lisibilité.

De manière générale, la lisibilité renvoie pour certains acteurs au fait de *comprendre, immédiatement* ou *très rapidement :*

> Donc voilà, pour moi ce qui est important, c'est que le code même de symfony, les gens puissent l'ouvrir, regarder, comprendre immédiatement (sf02).
>
> C'est un code où on comprend très rapidement, sans même lire mot à mot, ce que va faire l'ordinateur qui va exécuter le code (spip03).

L'un des acteurs rencontrés (sf06) distingue deux niveaux de lisibilité, l'un de « bas niveau », qui concerne la lisibilité syntaxique et l'autre, de « haut niveau », lié à la conception logicielle et à l'organisation du code source. Il nous semble intéressant de reprendre ces éléments ici.

Pour cet acteur, la lisibilité de bas niveau concerne d'abord des aspects comme l'indentation et l'espacement du code source : « ça va être bien indenté, avec des blocs bien définis » (sf06). Un autre acteur affirme quant à lui que « tu peux écrire tout ratatiné, donc c'est assez dur à lire. Tu peux écrire avec un peu plus d'espace, c'est plus facile à lire » (spip04). La figure suivante (5.1) illustre ces propos concernant la différence entre un code source « ratatiné » et un code source plus espacé, ou aéré.

---

115 <http://www.symfony-project.org/about> (consulté le 5 novembre 2011).

```
$res = '';
while($row=sql_fetch($result)){
    $id_breve=$row['id_breve'];
    if (autoriser('voir','breve',$id_breve)){
        $titre = typo($row['titre']);
        $statut = $row['statut'];
        $h = generer_url_ecrire('breves_voir',"id_breve=$id_breve");
        $res .= "<a class='$statut'
href='javascript:window.parent.location=\"$h\"'>$titre</a>";
        }
}
```

Le même code source, mais « ratatiné » :

```
$res='';while($row=sql_fetch($result))
{$id_breve=$row['id_breve'];if(autoriser('voir','breve',$id_breve))
{$titre=typo($row['titre'])$statut=$row['statut'];
$h=generer_url_ecrire('breves_voir',"id_breve=$id_breve");$res.="<a
class='$statut'href='javascript:window.parent.location=\"$h\"'>$titre
</a>";}}
```

Figure 5.1 : La lisibilité et l'indentation du code source
Code recopié de <http://core.spip.org/projects/spip/repository/entry/branches/spip-2.1/ecrire/exec/brouteur_frame.php> (consulté le 8 novembre 2011).

D'autres propos des acteurs vont également dans le même sens pour définir la lisibilité. Un acteur nous indique que la lisibilité, « ça va être un code aéré, avec des lignes qui vont découper les séquences d'instructions en séquences logiques » (spip11). Ce découpage des instructions en séquences d'instructions apparaît bien à la figure 5.1. Comme nous le mentionnerons plus loin, ce découpage est d'ailleurs l'une des conventions implicites les plus partagées parmi les informaticiens.

Le « bas niveau » de la lisibilité concerne également, selon notre interlocuteur, les noms de variables qui doivent être descriptifs : « quand je lis le nom d'une variable, il faut que je sache ce qu'il y a dedans, à quoi elle va servir » (sf06). En informatique, une variable est un élément du code source auquel on peut attribuer différentes valeurs, qui peuvent changer au cours du programme. L'extrait de code source présenté à la figure 5.1 inclut par exemple différentes variables identifiées par le premier caractère « $ » en PHP : *$row*, *$id_breve*, *$h*, *$res et $statut*. Certaines des variables présentées dans cette portion de code ont des noms plus descriptifs, tels que « $id_breve » et « $statut » dont on peut assumer qu'elles renvoient à *l'identifiant de la brève* et au *statut* d'un élément, vraisemblablement d'une brève. Cependant, la variable $h est, quant à elle, beaucoup moins descriptive : il est plus difficile de « comprendre immédiatement » ce que signifie cette variable sans lire le reste du code. Pour plusieurs acteurs, la question de la lisibilité renvoie à l'information que l'on peut tirer en lisant directement du code, sans nécessairement aller à la documentation. Ainsi, un acteur

mentionne en entrevue que s'il est nécessaire de consulter constamment la documentation, c'est que le code lui-même est illisible : « Si, au contraire, il faut que je passe des heures dans la doc, pour le moindre petit détail, etc., je vais dire que c'est illisible ! » (spip03). Les choix de nommage ont donc un caractère éminemment collectif, et ces choix doivent être régulés par différentes règles et conventions, ce que nous verrons plus loin dans le chapitre.

Si l'espace, l'indentation et le choix du nom des identifiants constituent le « bas niveau » de la lisibilité, la lisibilité de « haut niveau » concerne quant à elle le découpage adéquat du code source en différentes portions :

> Et après, pour que ce soit lisible à un plus haut niveau, il faut que le code soit aussi bien découpé. C'est-à-dire qu'on met des fonctions qui ont une taille bien délimitée.. Que le code soit justement découpé en fonctions, puis en classes, et puis en modules. Pour qu'on sache toujours un petit peu où aller (sf06).

Dans le chapitre précédent, nous avons déjà abordé ce découpage du code source en différentes portions. Nous avons ainsi mentionné un premier grand découpage, pour chacun des projets, entre le *core* et les *plugins,* ces derniers, au nombre de plusieurs centaines, constituant chacun des portions de code source – des « pièces de puzzle » – qui peuvent être articulées pour modifier le comportement du logiciel. Nous avons également présenté la manière dont, à l'intérieur du *core* et de chacun des *plugins*, le code source est également découpé en différents fichiers et répertoires. Dans la citation précédente, notre interlocuteur mentionne également que le code source doit être découpé « en fonctions, puis en classes, et puis en modules » (sf06), qui sont autant d'autres manières d'encapsuler le code source.

Pour plusieurs acteurs, ce découpage du code source en différentes portions est particulièrement important pour la lisibilité. Ainsi, notre interlocuteur note « qu'un fichier qui fait 5 000 lignes, ça n'aide pas forcément à se retrouver » (sf06). De la même manière, un autre acteur de symfony nous expliquait que « traditionnellement, dans les applications PHP on a un fichier dans lequel on a tout à l'intérieur. [...] Tout ça c'est mélangé. Ça fait des fichiers qui sont très longs, donc qui ne sont pas lisibles » (sf02). Même discours dans SPIP pour cet acteur qui fait référence aux propos de son beau-frère informaticien, pour qui « une fonction, ça ne doit pas dépasser une page à l'écran » (spip04).

Pour un des acteurs, un code lisible est aussi un code qui est écrit dans le style *habituel* :

> Mais, qu'est-ce que ça veut dire que c'est plus lisible, ça veut dire que, comme c'est le style habituel, c'est le style auquel je suis habitué, maintenant, j'ai moins besoin de réfléchir, et j'ai moins besoin de suivre en détail ce qui est écrit, pour comprendre ce que ça fait (spip01).

Cet acteur nous indique également qu'il privilégierait le choix d'un style habituel s'il avait à intégrer une nouvelle librairie, ou un nouveau morceau de code source dans son logiciel :

> Et je vais en chercher une [une librairie] qui non seulement fait le travail, mais si j'ai le choix je vais en choisir une où le style d'écriture est proche de celui dont je suis habitué. Ça veut dire que j'aurai plus de facilité à entrer dedans, à faire des modifications si besoin, à m'entendre avec l'auteur si il y a besoin de reporter les *patches*, ou quelque chose comme ça (spip01).

Pour cet acteur, le style d'écriture n'est pas seulement quelque chose d'interne à un projet, mais est également important pour la circulation des morceaux de code entre les différents projets. Un style commun dans un projet donné permet aux personnes de comprendre le code et d'interagir ensemble sur un même morceau de code source. Par conséquent, si chaque personne écrit dans son propre style, cela entraîne un manque de cohérence, qui rendra sinon illisible, du moins désagréable, la lecture du code. Bien que le terme « style » soit utilisé ici, il est important de noter que la notion de style utilisée ici est fortement normée. Ces styles d'écriture pourraient être comparables aux styles requis par les maisons d'édition ou par les journaux pour éditer les journaux. À la limite, cette notion de style de programmation se situerait à la frontière de la notion de style, comme celle « feuille de style » dans les logiciels de bureautiques, celle d'un « style » d'écriture ou d'un « style » artistique.

Nous voyons donc que le style et la lisibilité ne sont pas seulement une question interne à un projet. Elle permet également de mettre en commun différents projets. Ainsi, l'un des aspects intéressants des codes sous licence GPL est la possibilité de reprendre des morceaux de code d'un projet, pour les intégrer à un autre projet. Le style participe également à une certaine mise en compatibilité des différents morceaux de code. Ainsi, si les différents projets sont codés de manière très différente, le partage de parties du code source d'un projet à l'autre sera très difficile, voire contre-productif, puisqu'un morceau de code source au style très différent demandera en effet un travail important pour être intégré de façon élégante.

5.2.3 « Code propre », « code pourri », « code qui pue ». Autres catégories pour décrire la qualité du code source

D'autres catégories sont également mobilisées par les acteurs pour décrire la qualité interne du code source. Le contenu de ces catégories recoupe souvent celles présentées plus tôt (beauté, élégance, lisibilité). Ainsi, l'une des catégories souvent utilisées par les acteurs est celle de la propreté, de manière étroitement liée à celle de la lisibilité. Pour cet acteur de symfony qui avait décrit la lisibilité en deux niveaux, un code propre serait un code « qui est facilement lisible, qui est bien découpé, avec lequel tu peux interagir facilement, sans que ça créé des problèmes partout » (sf06). Un code propre serait également un code « découplé » : « c'est-à-dire que, pour chaque fonctionnalité, il y a une portion de code » (sf06). De la même manière, l'un des acteurs de symfony mentionne que « c'est pas propre dans le sens où si on veut modifier quelque chose, ben il y a plein de fichiers dans lesquels modifier » (sf02). Une actrice de SPIP mentionne quant à elle qu'un code propre est un code qui est bien documenté :

> Un code propre, c'est un code qui est documenté, qui est super bien documenté de manière à ce que, quand il y a des gens comme moi qui ne comprennent rien, et bien on arrive quand même à comprendre ce qu'a voulu faire le gars [sic] (spip08).

Un autre acteur (sf03) qui préfère quant à lui parler de l' « odeur du code », en parlant par exemple « du code qui sent bon, ou du code qui pue (sf03) ». Cet acteur note que l'odeur du code se perçoit avec l'expérience, par « une espèce d'intuition » (sf03), mais nous a quand même exposé certains critères d'un « code qui pue » : des fichiers qui font 3 000 lignes de long, des morceaux de code très imbriqués « quand il y a cinq niveaux d'imbrications, c'est du code qui pue ! » (sf03) et finalement, un mélange de plusieurs types de langage : « C'est tout à plat, c'est du mélange d'HTML, de PHP, de Javscript... Il y a plein d'éléments, plein d'odeurs! » (sf03). On retrouve encore ici une certaine idée de pureté qui s'exprime de façon inversée par le fait qu'un code pourri serait un code où l'on retrouve « plein d'éléments », un « mélange » de langages de programmation.

La présentation de ces différentes qualités permet de mettre de l'avant la manière dont le code source est organisé ou « découpé ». Elle rend compte du rôle de la qualité du code source, non seulement pour le bon fonctionnement des machines, mais également, et peut-être surtout, pour la collaboration entre les acteurs. La nécessité d'un code source de qualité est

généralement perçu par les acteurs comme un élément central de la collaboration, même si certaines qualités sont parfois contradictoires entre elles (par exemple, la beauté et l'ergonomie).

Le tableau suivant montre de façon succincte les différentes qualités du code source ainsi qu'une citation exprimant ce que chacune signifie. Dans la suite de chapitre, nous nous attardons plus précisément aux règles et aux conventions qui participent à la qualité du code source, ainsi qu'à la négociation par les acteurs de ces règles et conventions.

Tableau 5.1 : Extraits d'entrevue discutant de certaines qualités du code source

Qualités	Extraits d'entrevues
Beauté	« Un côté simple, épuration » (spip04) « Chaque fichier à sa place, et des fichiers courts » (spip04) « Très expressif, plutôt en un minimum de mots » (spip01) « Recherche de l'efficacité maximale avec la plus grande simplicité possible » (sf06) « Un peu inattendu, un peu astucieux » (spip01) « C'est vraiment malin » (sf04)
Élégance	« Ce découpage, la façon dont c'est écrit. La façon dont c'est aménagé » (spip06) « Bien présenté, c'est-à-dire que les commentaires sont en anglais, les variables sont en anglais, les variables sont intelligibles, l'indentation est respectée » (sf10) « Qu'il n'y ait pas trop de dépendances » (sf10)
Propreté	« Pour chaque fonctionnalité, il y a une portion de code » (sf06) « Intéressant de le lire, pour apprendre en fait » (sf09) « Un code qui est documenté, qui est super bien documenté » (spip08)
Lisibilité	« [Que] les gens puissent l'ouvrir, regarder, comprendre immédiatement » (sf02) « Un code où on comprend très rapidement, sans même lire mot à mot, ce que va faire l'ordinateur qui va exécuter le code » (spip03)
Illisible	« Un langage de programmation où les opérateurs font un seul caractère, chaque caractère, dans un certain ordre » (spip03) « Tu peux pas avoir une vue d'ensemble ! » (spip03) « Des fichiers qui sont très longs » (spip02)
Code pourri	« Pas de respect des conventions » (sf05)
Code qui pue	« Quand il y a 5 niveaux d'imbrications. [...]C'est tout à plat, c'est du mélange d'HTML, de PHP, de java script » (sf03)

5.3 Qualités du code source, conventions et bonnes pratiques

> Se conformer à l'utilisation de conventions, de standards de codage, de *design patterns*, de motifs de conception logicielle, ça permet de partager une culture commune... [...] Ça permet de faciliter la communication, en fait. Et a fortiori, la collaboration (sf08).

Dans cette partie, nous explorons les différentes règles qui favorisent une plus grande cohérence du code source. Nous présentons tout d'abord (5.3.1) quelques règles et normes de programmation explicites dans les projets, en nous attardant ensuite (5.3.2) plus spécifiquement aux conventions ayant trait au « nommage » (terme utilisé par les acteurs) de certaines parties du code source. Nous nous attardons ensuite (5.3.3) à la question du choix de l'anglais ou du français dans ces conventions d'écriture et de nommage, cette question linguistique étant un sujet sensible, du moins dans le cas de SPIP. La quatrième sous-section s'attarde davantage aux conventions et « bonnes pratiques » concernant la conception et l'organisation du code source. Finalement, nous terminons cette partie du chapitre en distinguant certaines conventions, que nous appelons « fortes », car elles sont contraintes par la machine, d'autres conventions que nous appelons « faibles », celles-ci ne reposant que sur la prescription humaine (5.3.5).

5.3.1 Règles, standards et conventions dans les projets étudiés

La fabrication collective du code source, dans les deux projets, s'appuie sur plusieurs règles et conventions dont l'énonciation prend différentes formes. D'emblée, mentionnons une différence importante entre SPIP et symfony quant à l'explicitation et au suivi de ces règles. Dans symfony, les règles sont beaucoup plus explicites et semblent suivies avec davantage de rigueur. On retrouve également dans ce projet un important discours sur les « bonnes pratiques », sur lesquels nous reviendrons à la section 5.3.4. Dans SPIP, au contraire, plusieurs acteurs que nous avons rencontrés en entrevue déplorent l'absence de règles :

> On a besoin de trouver quelque chose comme ça dans SPIP, et on n'a pas trouvé. Chacun écrit son code comme il veut. Après il y a le problème des noms. Pendant longtemps, on n'utilisait que des mots français. Après les nouvelles recrues ont utilisé des mots anglais, ça fait un truc vraiment bordélique ! (spip03)

> Dans SPIP, il n'y a aucune règle, aucune directive. Du coup, chacun code comme il veut, chacun nomme ses fonctions comme il veut (spip09).

Cette perspective d'une absence complète de règles dans SPIP doit cependant être nuancée. L'introduction de la page « Contribuer au développement de SPIP » explique par exemple que « Ce projet est muni d'un ensemble de règles qui, toutes arbitraires qu'elles peuvent paraître, assurent sa cohérence[116] », puis énonce certaines règles précises dont nous présenterons quelques extraits plus loin. Selon quelques acteurs que nous avons rencontrés, cette page est cependant datée et ces règles ne s'appliqueraient plus aujourd'hui. Bien que notre enquête ne permet pas de confirmer, par l'observation, la manière exacte dont les règles sont appliquées, une chose est certaine, SPIP se distingue de façon importante de symfony par un important « libéralisme » concernant l'uniformisation de l'écriture du code source.

Les deux projets présentent chacun une page décrivant certaines règles (bien que, comme mentionné plus tôt, il n'est pas certain que cette page soit toujours en vigueur dans SPIP). Dans les deux projets, ces règles sont décrites sur la page décrivant comment contribuer. Dans SPIP, la page « Contribuer au développement de SPIP » (mentionnée plus tôt) présente une section intitulée « Règles de présentation et d'écriture » où sont stipulées ces quelques règles. Dans symfony, la page « How to contribute to symfony » inclut quant à elle une section intitulée « Coding standards ». Mentionnons par ailleurs que l'introduction de cette section (dans symfony) renvoie à la page Wikipédia des « Coding standards[117] ».

Avant de présenter plus en détail ces règles d'écriture et standards de codage, mentionnons que les deux projets attachent une grande importance à ce que le projet SPIP appelle des règles « tacites », qui doivent également être respectées :

> Ces règles n'ont pas besoin d'être énoncées explicitement pour exister : certaines sont clairement visibles après un examen plus ou moins détaillé du code, et les règles tacites doivent être respectées au même titre que les autres (site web de SPIP[118]).

En préambule de ses règles, la page web de symfony spécifie quant à elle que la règle d'or est d'imiter le code existant : « Here's the golden rule : Imitate the existing symfony code ».

---

116 <http://www.spip.net/fr_article825.html> (consulté le 23 novembre 2011).

117 <http://en.wikipedia.org/wiki/Coding_standards> (consulté le 24 novembre 2011).

118 <http://www.spip.net/fr_article825.html> (consulté le 23 novembre 2011).

Le suivi des règles d'écriture et des conventions de nommage, dans SPIP, mais surtout dans symfony, est fortement valorisé. Ce suivi des règles participe à la propreté, la lisibilité, voire à une certaine beauté du code source, qualités du code source qui participent elles-mêmes à un certain respect pour les autres contributeurs. Une des actrices de symfony va cependant plus loin en affirmant que le suivi des règles et des conventions (dans ce cas-ci, la question de la langue du code source) constitue en quelque sorte un certain contrat moral :

> Mais je considère si tu utilises symfony, ou Zend Framework, ou CakePHP, ou n'importe quel outil... par exemple, si j'utilise SPIP, je vais faire en français, parce que j'ai accepté un contrat, finalement, de continuer à coder, euh, pareil. Donc, si j'utilise symfony, j'ai accepté le contrat que j'allais mettre mes noms de fonction en anglais et mes noms de variables en anglais. [...] Donc, pour moi, c'est juste quand tu utilises un outil, tu passes un certain contrat, euh, moral, hein (sf05).

De manière générale, les règles d'écriture et les standards de codage concernent avant tout des aspects « stylistiques » de l'écriture du code source. Ces conventions ont trait par exemple à l'espacement et à l'indentation, ou à la manière de poser les accolades. Sans présenter chacune de ces règles, mentionnons par exemple les règles dans chacun des projets qui ont trait à l'indentation du code source. Ainsi, dans SPIP, l'indentation doit être faite avec le caractère de tabulation :

> L'indentation sera faite de préférence avec le caractère de tabulation. Cela permet de choisir librement la profondeur d'indentation dans les options de son éditeur de texte, tout en n'imposant pas ce choix aux autres développeurs.[119]

Dans symfony, comme on le voit à la figure 5.2, l'emploi du caractère de tabulation est au contraire proscrite :

119 <http://www.spip.net/fr_article825.html> (consulté le 23 novembre 2011).

Never use tabulations in the code. Indentation is done by steps of **2 spaces**:

```
<?php
class sfFoo
{
  public function bar()
  {
    sfCoffee::make();
  }
}
```

Figure 5.2 : Une règle stylistique : ne pas utiliser de tabulations
<http://trac.symfony-project.org/wiki/HowToContributeToSymfony> (consulté le 24 septembre 2011).

Nous pouvons ici rejoindre les propos de Denis et Pontille (2010a) dans leur analyse de la performativité des artefacts écrits. Faisant référence à la théorie de l'acteur-réseau, les auteurs notent que la performativité des artefacts repose en bonne partie sur la stabilité des relations qu'ils entretiennent dans un réseau, stabilité qui est également une condition de la circulation de ces artefacts. Citant Latour, ils notent que « la force des artefacts est conditionnée à leur capacité à devenir des « mobiles immuables » (Latour, 1985), c'est-à-dire des objets qui peuvent circuler d'un point à un autre du réseau sans changer d'état, ni perdre leur forme » (Denis et Pontille, 2010a). Dans le cas du code source, les règles et les conventions d'écriture favorisent la circulation d'un morceau de code source, à la fois à l'intérieur d'un projet, mais aussi entre les différents projets. C'est donc en quelque sorte le respect des règles d'écriture qui donne au code une certaine stabilité pour pouvoir être articulé dans un réseau plus large.

5.3.2 Les conventions de nommage

Une partie des règles d'écriture peuvent être décrites comme des « conventions de nommage », une expression utilisée par les acteurs. Ces conventions de nommage sont des règles qui concernent la manière de nommer des identifiants, par exemple le nom des fichiers, le nom des variables ou des fonctions dans le code. Les règles et standards de chacun des projets stipulent certaines règles de base concernant les projets. La page « Contribuer au développement de SPIP » stipule ainsi que :

> Quel que soit le projet, le nommage doit rester homogène pour que le code soit facile à lire. Ainsi, sous SPIP, les noms de variables et de fonctions seront en minuscules ; les noms composés, de la forme variable_composee[120].

La figure suivante (5.3), issue de la page « How to contribute to symfony », expose également une règle concernant les noms de variables, de fonctions ou de méthodes[121] dans symfony. Mentionnons encore une fois la différence entre SPIP, qui préconise les minuscules et les noms composés, et symfony, qui préconise les majuscules pour séparer les mots et proscrit les noms composés :

```
Use camelCase, not underscores, for variable,
function and method names:
    • Good: function makeCoffee()
    • Bad: function MakeCoffee()
    • Bad: function make_coffee()
```

Figure 5.3 : Règle préconisant l'utilisation de la majuscule pour le nommage
<http://trac.symfony-project.org/wiki/HowToContributeToSymfony> (consulté le 24 septembre 2011)

Le nommage renvoie principalement, comme le note la figure précédente aux noms de variables, de fonctions, ou de méthodes, mais il peut aborder plusieurs autres aspects. Deissenboeck et Pizka (2006) notent ainsi que 70% du code source d'un logiciel consiste en des identifiants[122]. Le nom des fichiers est ainsi l'un des éléments concernés par la question du nommage (figure 5.4, page suivante).

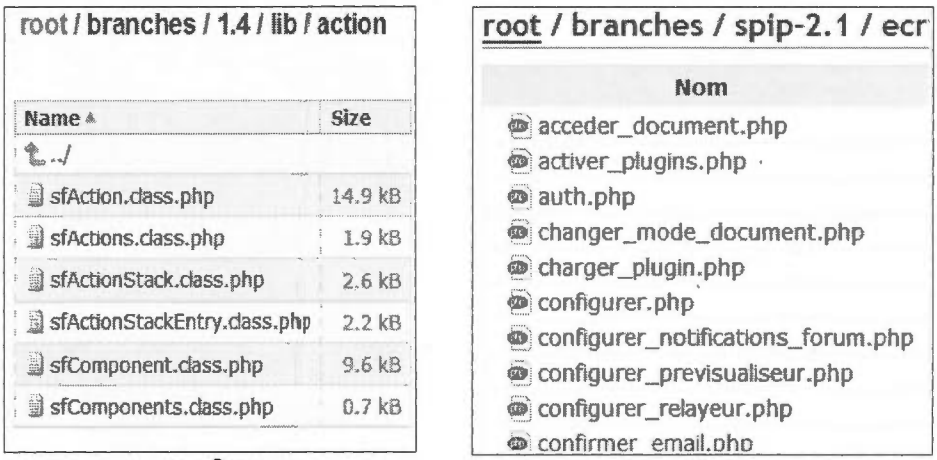
Dans le cadre des deux projets, cette question du *nommage* revient très souvent et l'expression « conventions de nommage » ou « naming conventions » est fréquemment

---

120 <http://www.spip.net/fr_article825.html> (consulté le 23 novembre 2011).

121 Dans le cas de PHP, les *variables* sont des composantes logicielles qui sont utilisées pour décrire une composante variable. Les *fonctions* ou les *méthodes* sont des portions de code source encapsulées ou isolées dans des procédures abstraites.

122 Étudiant le cas du logiciel Eclipse (3.1.1), les auteurs notent l'existence de 142 275 noms différents dans le code source de ce logiciel, ce qui dépasse le nombre de mots dans le dictionnaire *Oxford Advanced Learner's Dictionary* (Deissenboeck et Pizka, 2006, p. 274). Ils remarquent également que les conventions de nommage sont la plupart du temps très peu explicites dans les projets de développement de logiciels, et proposent l'utilisation d'un dictionnaire d'identifiants (*identifier dictionary*) pour faciliter le choix des noms des identifiants du code source.



symfony **SPIP**

Figure 5.4 : Noms des fichiers dans symfony et SPIP
<http://trac.symfony-project.org/browser/branches/1.4/lib/action> (consulté le 9 novembre 2011) et <http://core.spip.org/projects/spip/repository/show/branches/spip-2.1/ecrire/action> (consulté le 24 novembre 2011).

utilisée pour désigner les différentes règles de nomenclature concernant le nom des fichiers ou de certaines parties du code source[123]. La question du nommage renvoie notamment à celle de la lisibilité. D'une part, nous l'avons mentionné, si un nom de fonction n'est pas compréhensible, il est difficile pour les autres de le lire. Le nom d'une fonction doit donc pouvoir informer rapidement le lecteur du code. En fait, le nom même d'un objet doit pouvoir de lui-même comporter suffisamment d'informations sur le comportement de l'objet en question. La question du nommage est une question importante, autant pour la communication entre les concepteurs que pour la « communication » entre les différents morceaux de code.

Dans le cadre de SPIP, au contraire, les conventions sont pour ainsi dire très souples et elles n'ont pas été établies a priori :

123 Wikipédia définit par exemple le terme de convention de nommage comme « un ensemble de règles destinées à choisir la séquence de caractères à utiliser pour les identifiants dans le code source et la documentation ».
<http://fr.wikipedia.org/wiki/Convention_de_nommage> (consulté le 9 novembre 2011).

> Au début, ils étaient trois, chacun travaillait sur une partie, il n'y avait pas trop d'interactions. [...] Mais de nouveaux contributeurs sont arrivés, et puis, il n'y avaient pas forcément des conventions de posée, donc euh, chacun y allait un peu au *feeling*, en fonction de ses habitudes à lui. Et au fil du temps, on se retrouve avec des grosses hétérogénéités sur le nommage, des fois c'est un verbe des fois c'est en français, des fois c'est en anglais (spip11).

Les choix et les conventions de nommage renvoient aussi à des dynamiques de reconnaissances institutionnelles. Ainsi, dans symfony, l'une des règles importantes est de préfixer le nom des « classes » et des fichiers par les lettres *sf*, qui signifie *s*ym*f*ony, mais également ***S***ensio ***F***ramework, *Sensio* étant le nom de l'entreprise qui mène le projet[124]. À ce titre, mentionnons une conversation[125] intéressante qui s'est déroulée au tout début de notre enquête, et qui concernait la convention à adopter pour le préfixe des noms des plugins. Rappelons que les plugins sont des fonctionnalités, ou des bouts de code source, qui sont développés par des acteurs externes à la *core team*. L'origine de la conversation consistait à choisir, pour le nom du plugin, entre ***sf***ChatPlugin et ***ll***ChatPlugin. Le premier préfixe – sf – renvoyant à la convention standard et le second – ll – renvoyant au nom de l'entreprise qui l'avait développé et qui souhaitait conserver une référence à son projet. L'analyse de cette conversation permet de faire ressortir différents régimes de justification du choix des noms de préfixe. Tout d'abord, certains acteurs sont favorables à l'idée de laisser la possibilité à l'entreprise d'inscrire une référence à sa marque au sein même du code source, de façon à favoriser la contribution des entreprises dans symfony « Everyone has to *think about branding* to some extent »[126]. D'autres rejettent cette idée sous prétexte que de publier un plugin au nom de l'entreprise affaiblit la collaboration dans le projet : « If other developers join your team, why should they develop a plugin that is published in the name of your

---

124 Un aspect intéressant à noter ici est que c'est le préfixe *sf* qui est à l'origine du nom symfony, et non l'inverse. En effet, l'entreprise qui commandite le projet – Sensio – avait initialement développé un framework interne nommé Sensio Framework, et dont le préfixe de plusieurs des identifiants du code source était SF. L'entreprise décida ensuite de rendre public son logiciel ce qui impliquait de trouver un autre nom moins lié à l'entreprise. Plutôt que de réécrire le code source, ils décidèrent de choisir un nom qui reflétait les préfixes utilisés dans le code source, soit *s*ym*f*ony.

125 Comme mentionné dans le chapitre méthodologique, le terme « conversation » dans notre étude renvoie au terme utilisé par Google, ou Thunderbird et décrit une suite de courriels se répondant l'un à l'autre. La conversation dont il est question ici s'est déroulée sur une période quatre jours, en février 2009, et a suscité au total 16 interventions (sous forme de courriels envoyés à la liste).

126 Nous soulignons.

company? Such an attitude only *weakens collaboration* »[127]. D'autres encore rejettent également cette proposition de préfixe spécifique à l'entreprise sous prétexte que le choix de différents préfixes pour le nom des plugins créera de la confusion dans le rôle des différents plugins et nuira à la qualité du code source. Un acteur remarque ainsi qu'il existe plusieurs plugins aux rôles similaires, mais aux noms différents, tels que *sfSimpleCMSPlugin*, *sfSimpleCMS2Plugin*, *sfInstantCMSPlugin*, situation qui pourrait devenir encore pire avec des noms de plugins tels que *pkSimpleCMSPlugin*, *abSimpleCMSPlugin*, *otSimpleCMSPlugin*. Notons que cette situation se complique également du fait que plusieurs des plugins publiés sur le site de symfony sont dans un état douteux, certains étant très bogués, voire tout simplement dysfonctionnels. Certains acteurs souhaitent donc que soient distingués plus clairement les plugins fonctionnels et bien maintenus, d'autres plugins potentiellement abandonnés.

Une solution de compromis est ressortie de cette discussion. D'une part, certains ont proposé qu'il y ait des « core plugins » auxquels serait attribué le préfixe sf. Ces plugins seraient donc reconnus par l'équipe de « core » de symfony, et le préfixe sf serait en quelque sorte un gage de la qualité du plugin. Les autres plugins pourraient quant à eux utiliser d'autres préfixes, par exemple ceux des entreprises. D'autre part, le seul choix des préfixes ne pouvant pas tout dire de la qualité d'un plugin, il a été suggéré que le dépôt des plugins soit amélioré de façon à pouvoir mieux évaluer et présenter la qualité de chacun de ceux-ci.

5.3.3 Anglais ou français : la langue du code source

Liée à la question du nommage et de la lisibilité, la question de la langue du code source est particulièrement importante dans les deux projets, bien que celle-ci soit considérée comme une question réglée dans le cas de symfony. Nous l'avons déjà dit dans le chapitre de méthodologie, symfony est un projet résolument tourné vers le monde anglophone, et la plupart des sites de symfony, en tout cas, les sites officiels, sont rédigés en anglais. Pour justifier ce choix de l'anglais, un acteur nous donne d'ailleurs l'exemple de SPIP, qui est très connu en France, mais qui, selon lui, n'est pas très connu ailleurs dans le monde. Dans le cas de symfony, l'objectif était plutôt, dès le départ, de rejoindre une communauté internationale

127 Nous soulignons.

et le choix de l'anglais a été important puisque, selon cet acteur : « La seule langue informatique qui existe dans le monde, c'est l'anglais » (sf02). Cet acteur donne l'exemple du langage *Ruby*, un langage réalisé d'abord par les Japonnais et dont la documentation était également en japonnais, ce qui fait que Ruby était très utilisé au Japon, mais à peu près méconnu ailleurs dans le monde. Ce n'est qu'avec l'arrivée de *Ruby on Rails*, un autre logiciel du type Framework, mais basé sur Ruby, et l'émergence d'une documentation anglophone, que Ruby s'est fait connaître à l'extérieur du Japon. La conclusion de cet acteur sur la nécessité d'utiliser l'anglais dans les projets open source est claire et percutante :

> Je pense qu'aujourd'hui, on ne peut pas décemment vouloir créer un logiciel open source, dans une autre langue que l'anglais. Si on veut avoir une portée mondiale, si on veut avoir un vrai rayonnement, c'est pas possible (sf02).

De la même manière, un autre acteur mentionne qu'il faut écrire la documentation en anglais, sans quoi symfony « n'aurait aucune chance d'exister avec un volume important de contributions » (sf03). Pour un autre acteur, il s'agit d'une « bonne pratique » :

> C'est que moi, la langue de l'informatique, c'est l'anglais. Donc, dans les logiciels, tout doit être en anglais. Donc, le code source doit être en anglais, les commentaires doivent en anglais, les messages de commit doivent être en anglais. C'est mon opinion, pour moi c'est une bonne pratique. [...] Quelque part, normalement, toutes les personnes qui développent devraient parler en anglais. Pour moi, c'est une condition sine qua non pour l'intervention dans l'univers informatique. Ça permet d'avoir une langue qui est commune à tous les développeurs (sf06).

Cependant, même dans le cas de symfony, la question de la langue peut être problématique puisqu'il ne s'agit pas de la langue « native » des gens, ce qui peut créer quelques incompréhensions :

> Ça veut dire que de temps en temps, on peut écrire des choses dont on pense qu'elles ont une certaine signification, une certaine portée, mais en fait, pour un anglophone, elles ont une portée qui est complètement différente (sf02).

Dans le cas de SPIP, la question de la langue du code source est plus problématique. D'abord, notons que le code source de SPIP est écrit d'une manière qui hybride certains mots anglais et certains mots français (voir par exemple la figure 5.1). Cette situation n'est pas le résultat d'un choix en tant que tel, mais est plutôt le reflet d'une évolution organique du code source, à partir des premiers scripts « personnels » qui avaient été écrits en français et qui ont ensuite été réunis dans SPIP. Pour reprendre les termes de Suchman (1987; 2007), le langage SPIP –

ou la langue du code - n'a pas été choisi « par plan », mais qu'elle a plutôt émergé en situation : les gens ont écrit quelques scripts en français et ces scripts sont devenus plus tard un produit aux frontières plus définies.

Cependant, la langue du code source de SPIP reste aujourd'hui une question débattue dans SPIP. D'abord, selon les termes d'un acteur, on peut remarquer la présence d'acteurs « de tendance historique qui sont super attachés à ce qu'on reste bien dans la langue française et qui font super gaffe à ça » (spip09). Ainsi, un des acteurs que nous avons rencontrés justifie que le code source de SPIP soit en français par le fait que cela favorise l'inventivité au sein d'une communauté francophone :

> Je suis persuadé qu'on invente dans sa langue, l'imagination, les idées nouvelles... À part si on est vraiment bilingue. S'imposer le véhicule d'une langue dans laquelle on n'est pas complètement à l'aise est un frein à l'inventivité. J'en suis persuadé ! (sf03)

De façon générale, c'est donc surtout pour des raisons cognitives ou sociales que les gens privilégient l'emploi du français[128]. Certains, surtout des acteurs en position plus périphérique dans le projet, indiquent ainsi que le fait que SPIP soit en français, et en particulier que la documentation soit en français, facilite leur compréhension du code de SPIP :

> Donc, dès l'instant que c'est français, déjà c'est beaucoup plus rapide, plus simple. Toute la documentation, pareil, on la trouve en français. Il y a une grosse communauté francophone qui l'utilise aussi. Sur les listes de diffusion, c'est quasiment tout en français, donc c'est relativement pratique (spip02).
>
> Je suis tombé sur SPIP un peu par hasard, et en même temps, pas tout à fait par hasard, puisqu'il y a tout de suite eu un truc qui m'accrochait, parce que je suis Français francophone, c'est évidemment que la documentation était en français, parce que c'est écrit par des Français [...]. Même, si à partir du moment où tu codes, tu entres dans cette espèce de jargon anglo-codeur, on va dire. Il n'empêche qu'il y a subitement un tas de finesse à cause de la documentation en français (spip07)[129].

---

128 Dans les entrevues que nous avons réalisées, ce sont surtout des arguments cognitifs ou sociaux qui sont mis de l'avant pour justifier le choix du français dans la communauté. Nous n'avons cependant pas rencontré des arguments plus « identitaires » justifiant l'usage du français, comme ce serait par exemple le cas au Québec.

129 Ces extraits d'entrevues montrent, comme nous l'avons exposé au chapitre 4, les liens étroits entre le code source et la documentation et, dans ce cas-ci, le fait que la langue de la documentation est intimement liée à celle du code source.

Cette préférence pour l'usage du français ne fait cependant pas l'unanimité parmi les acteurs que nous avons rencontrés. D'autres acteurs regrettent ainsi que le code source soit écrit en français, car cela a pour effet de couper la communauté du milieu anglophone :

> Le choix des termes « boucles, balises », ça ne veut rien dire pour les anglophones. L'effort d'apprentissage est tellement important, que je ne m'y mettrais pas. [...] Les codeurs vont difficilement aller dans le code, à cause du nommage des fonctions. L'anglais en informatique, c'est la langue universelle. [...] SPIP, c'est pas une communauté d'informaticiens, alors c'est pas dans leur priorité. Il y en a qui sont allergiques à l'anglais. Le fait de mettre des commentaires en anglais dans le code, le nommage des fonctions. Tous les commentaires sont en français. J'ai vraiment du mal à me mettre à la place des anglais (spip09).

Lié à la question de l'usage du français, un autre problème dans SPIP a trait au manque d'harmonie qu'implique l'usage combiné de l'anglais et du français. Plusieurs acteurs mentionnent le caractère rebutant de cette question :

> Le code, par contre, il m'a un peu rebuté, car il intégrait des mots en français, c'était bizarre au départ, car c'est vraiment inhabituel (spip07).

Un autre acteur nous mentionne toutefois n'éprouver aucun remord personnel à hybrider le français et l'anglais, tout en notant les « horreurs linguistiques » qui peuvent découler de cette approche :

> Enfin, personnellement, je n'ai pas de remords à utiliser le nom anglais, collé à un début de nom français parce qu'au global, ça fait un truc plus court et, voilà, c'est plus rapide à écrire. Ça reste compréhensible quand même, etc. Mais des fois, ça produit des horreurs linguistiques (spip11).

La question de l'hybridation et de la normalisation linguistique est récurrente dans le projet SPIP. Ainsi, une des questions sur laquelle les acteurs ont préféré s'entendre est sur l'utilisation de l'anglais concerne les fonctions SET et GET. Une discussion a eu lieu pour utiliser des termes francophones équivalents. Cependant, les termes SET et GET étant très connus pour les acteurs du web et puisqu'il était difficile de trouver une formulation aussi courte pour signifier la même action, il a été décidé de s'en tenir aux termes GET et SET :

> Le SET et le GET, c'est quasiment une condition dans les langages informatiques et du coup, ça n'a aucune ambiguïté. Alors que quand tu commences avec METTRE et PRENDRE en fait, tu sais pas de quel côté il faut les placer, de quel côté il faut les comprendre. Ça peut marcher dans les deux cas, et comme ce n'est pas une convention, ça porte à ambiguïté du fait que c'est pas un usage répandu. [...] En fait, ce qui s'est passé c'est que j'ai mis SET et GET, là je me suis fait trollé[130] parce que c'était en anglais. On en a discuté pour savoir, on a cherché une discussion. Et au final on a conclu collectivement que c'était bien dommage, mais qu'on n'arrivait pas à faire mieux quoi (spip09).

5.3.4 « Bonnes pratiques » et motifs de conception dans symfony

Au sein de symfony, contrairement à SPIP, on retrouve un discours important sur les « bonnes pratiques » ou en anglais, les « best practices ». La référence constante aux bonnes pratiques semble d'ailleurs être un argument de vente. Ainsi, la page principale du projet insiste sur le fait que symfony utilise la plupart des bonnes pratiques du développement web :

> Symfony is based on experience. It does not reinvent the wheel: it uses most of the best practices of web development and integrates some great third-party libraries[131].

À quoi renvoient ces bonnes pratiques ? Selon les acteurs que nous avons rencontrés, ces bonnes pratiques, prises dans un sens général, renvoient à un large spectre de pratiques. Ainsi, pour une des actrices (sf05), le fait d'indenter son code et de le mettre sur plusieurs lignes, plutôt que sur une seule, constitue une « bonne pratique » facilitant la lisibilité. Selon cette actrice, les bonnes pratiques sont davantage liées au travail en équipe, au fait de produire du code qui soit pérenne et qui facilite la collaboration. Un autre acteur décrit ainsi l'éventail des pratiques auxquelles renvoie le terme « bonne pratique » :

---

130 Un *troll* est un message provocateur (par exemple sur une liste de discussion) visant à susciter plusieurs réactions ou à enflammer un débat. Dans le contexte de cette citation, *se faire troller* signifie donc de recevoir une multitude de messages courriels visant un débat sur cette question. Dans SPIP (mais moins dans symfony), la pratique du trollage, consistant en de longues discussions sur les listes de courriels, est assez courante et acceptée.

131 <http://www.symfony-project.org> (consulté le 6 novembre 2011).

> Les bonnes pratiques, il y en a plein, il y a en à plein de niveaux. Ça va d'une bonne pratique de documenter, ça a rien à voir avec le code. Jusqu'à, c'est une bonne pratique de tester, qui est plutôt un concept. Jusqu'à une bonne pratique, c'est de faire du MVC[132], où là c'est vraiment du code, ou de l'organisation du code (sf02).

D'où viennent ces bonnes pratiques ? Plusieurs acteurs nous indiquent que ces bonnes pratiques viennent de l'expérience : « Ceux qui font les programmes se sont aperçu que c'est plus pratique de structurer son code en modules, de séparer la couche présentation, logique, règles de métier » (sf03). Parfois, ces pratiques proviennent de l'expérience personnelle de développeurs séniors dans symfony et sont diffusées lors de conférences ou sur le web. Certaines bonnes pratiques ont une valeur plus « canonique » et proviennent de développeurs connus sur le site web.

L'un des « horizons paradigmatiques » des bonnes pratiques concerne les « méthodes agiles ». Il nous semble important de présenter quelques mots à propos de ces méthodes. Barcellini (2008) définit dans sa thèse les méthodes agiles comme un « ensemble de méthodes visant à simplifier le processus de conception logicielle et à donner une place importante aux clients dans ce processus » (p. 17). Elle note également que ces méthodes « prônent le prototypage rapide et la mise en production rapide des modifications » (p. 17). Les méthodes agiles ont émergé vers la fin des années 1990 et ont été officialisées en 2001 par le Manifeste Agile (*Agile Manifesto*). Ce document a été signé par 17 développeurs de logiciels, dont l'un des plus connus est sans doute Ward Cunningham, le créateur du wiki[133]. Ce manifeste spécifie différentes « valeurs fondamentales », dont l'interaction avec les personnes, la valorisation d'un produit opérationnel plutôt qu'une documentation exhaustive, la collaboration avec le client plutôt que la négociation du contrat et l'adaptation au changement plutôt que le suivi d'un plan[134].

Au sein de symfony, les références aux principes agiles restent la plupart du temps implicites. Ainsi, dans le cadre de nos entrevues, seuls deux acteurs font mention de ces méthodologies

---

132 MVC : *Model View Controller*. La signification de ce terme sera abordée plus loin dans cette section.

133 Voir Goldenberg (2010, p. 21) pour une description du premier wiki.

134 <http://agilemanifesto.org/iso/fr/> (consulté 15 novembre 2012).

et dans le second cas, c'est nous-mêmes qui avons questionné l'acteur à ce propos[135]. Ce sont plutôt les principes ou les méthodes particulières de la méthodologie agile qui sont mobilisés par les acteurs. L'ancien site web de symfony réfère cependant explicitement aux principes de développement agile. Sur la page *About*, il est écrit que « Developers can apply agile development principles (such as DRY, KISS or the XP philosophy[136]) and focus on applicative logic without losing time to write endless XML configuration files »[137]. Un tutoriel publié en décembre 2008 est également présenté comme ayant pour objectif d'illustrer le développement agile avec le framework symfony (« to illustrate agile development of a web application in PHP with the symfony framework[138] »).

Un autre « horizon paradigmatique » concerne cette fois-ci les « motifs de conception » ou pour employer le terme anglophone beaucoup plus utilisée parmi les acteurs, les *design patterns*. Gamma et al. (1994), qui ont popularisé cette expression dans le domaine du développement de logiciels, définissent les *design patterns* comme des « descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context » (p. 13). Les *design patterns* ne sont donc pas des morceaux de code, ou une conception finalisée, mais plutôt des modèles de conception généraux qui doivent ensuite être adaptés à des situations particulières. En français, les *design patterns* sont traduits par « motifs de conception » ou « patrons de conception ». Les *design patterns* concernent les relations entre différentes composantes du code source; ils se situent surtout au niveau de l'organisation du code source. Ainsi, le MVC, qui a été mentionné dans un extrait précédent d'entrevue, est l'un de ces *design patterns* particulièrement mobilisé dans le cadre de symfony (nous verrons que ce *design pattern* est également mobilisé dans le cadre de SPIP, bien que

---

135 C'est à la suite d'une rencontre avec Françoise Détienne, qui a réalisé plusieurs études sur la psychologie de la programmation et les communautés de logiciels libres (Barcellini et al., 2006; Détienne, 1998; Barcellini, Détienne, et Burkhardt, 2007), que nous avons entrepris de questionner quelques acteurs sur les principes agiles. Par la suite, l'analyse a également permis de faire ressortir que les méthodes agiles avaient été citées dans une de nos premières entrevues, et que certaines pages web du projet symfony y faisaient référence.

136 DRY : Don't repeat yourself (Ne vous répétez pas); KISS : Keep it simple, stupid (gardez ça simple, stupide); XP : eXtreme Programming.

137 <http://www.symfony-project.org/about> (consulté le 5 novembre 2011). La nouvelle version de cette page (consultée le 9 novembre 2011) n'aborde cependant pas la question des principes du développement agile.

138 <http://www.jobeet.org/about> (consulté le 17 novembre 2011).

de manière plus implicite). MVC est un acronyme qui signifie Modèle Vue Contrôleur (ou en anglais : *Model View Controller*). Il s'agit d'un principe d'organisation du code qui préconise de le séparer dans différents fichiers constitutifs du code source :

> Donc entre autres, il y a un *pattern*, qu'on appelle le *pattern* MVC, modèle vue contrôleur, et qui donne une manière élégante de gérer des projets un peu compliqués. En isolant la partie modèle, la partie métier, de la partie vue, qui est la partie vraiment représentation, de la partie contrôleur qui est la partie aiguillage, d'interaction avec l'utilisateur [...]. Cette méthodologie-là, elle permet entre autres d'éviter le code spaghetti, c'est-à-dire de tout mélanger, et d'avoir une séparation de responsabilités sur ces trois couches (sf10).

Comme pour les méthodologies agiles, il est encore utile d'inscrire les *design patterns* dans le cadre d'un mouvement plus large et historique. On retrouve par exemple une référence directe aux *design patterns* dans le titre de l'article séminal de O'Reilly sur le web 2.0 (O'Reilly, 2007). Il est également frappant de constater que ces bonnes pratiques et ces d*esign patterns* trouvent leur matrice historique chez des auteurs qui sont également à l'origine du wiki. Le concept de *design pattern* trouve en effet son origine dans le domaine de l'architecture, dans les travaux de Christopher Alexander (1977). Vers la fin des années 1980, Kent Beck et Ward Cunningham (1987) ont également commencé à expérimenter avec l'idée de *design pattern* dans le domaine de la programmation. Puis, dans le milieu des années 1990, les *design patterns* sont devenus plus populaires avec la publication de l'ouvrage, mentionné plus tôt, *Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma et al., 1994). La même année (1994) a eu lieu la première conférence sur les *design patterns*[139], puis l'année suivante, le site web *Portland Pattern Repository*[140], qui visait à documenter et collectionner les *design patterns*, a été lancé[141]. Créé par Ward Cunningham, ce site web constitue d'ailleurs le premier wiki historique (Goldenberg, 2010, p. 28). Cette présentation de l'origine des d*esign patterns* montre bien les liens étroits qui existent entre le code source et d'autres médias de type « web 2.0 », notamment les wikis[142]. Le code source doit non seulement être

---

139 <http://hillside.net/plop/plop94/> (consulté le 28 février 2012)

140 <http://c2.com/ppr/> (consulté le 28 février 2012)

141 Le bref historique des *Design Patterns* décrit dans les dernières lignes est basé sur Wikipédia : <http://en.wikipedia.org/wiki/Software_design_pattern#History> (consulté le 28 février 2012).

142 D'autres auteurs ont ensuite marqué le mouvement des *design patterns*, en particulier Martin Fowler, dont l'ouvrage *Patterns for Entreprise Applications (Fowler, 2003)* est considéré comme un « Best Seller » par l'un des acteurs (sf03).

appréhendé comme un artefact de la fabrication de ces médias, mais les pratiques d'écriture et d'organisation du code source semblent elles-mêmes constituer une matrice d'expérimentation de pratiques du « web 2.0 ».

5.3.5 Conventions faibles et conventions fortes

Parmi les différentes règles et conventions, il est intéressant de distinguer entre ce que nous appelons ici les conventions « faibles » et les conventions « fortes ».

Tout d'abord, les conventions qui pourraient être qualifiées de « faibles », sont des conventions proprement dites, qui n'ont aucun impact sur le fonctionnement de la machine et qui ne visent qu'à la collaboration entre les acteurs. La lisibilité, surtout lorsqu'elle concerne l'espace et l'indentation, est un aspect intéressant de notre étude, car il s'agit d'une qualité du code source qui renvoie à un aspect de communication entre les humains et qui n'a absolument rien à voir avec le fonctionnement de la machine. L'espacement du code source sert uniquement à faciliter la lecture par les humains. En effet, comme nous l'avons montré à la figure 5.1, deux portions de code source auront le même résultat en termes de fonctionnement de l'ordinateur, même si l'une est « ratatinée » et que l'autre est bien indentée et davantage lisible[143].

Ensuite, des conventions que nous pourrions appeler « fortes », qui ne visent pas uniquement à une meilleure lecture, ou une meilleure collaboration, mais sont également obligatoires au fonctionnement du programme (ou de la machine) : si ces conventions ne sont pas respectées, le programme ne fonctionnera pas correctement. L'identifiant '$' au début des variables, que nous avons abordé plus tôt, est un bon exemple de convention « forte » où le programme ne fonctionnera pas correctement si elle n'est pas respectée. Dans ce cas-ci, la variable ne sera simplement pas reconnue en tant que telle par la machine.

Le langage PHP, au contraire d'autres langages de programmation, permet de plus aux usagers (les programmeurs qui utilisent PHP) de développer leurs propres conventions fortes.

---

143 Bien que ce soit le cas dans le langage PHP, ce n'est pas nécessairement le cas dans d'autres langages de programmation. Un acteur (sf10) nous mentionne par exemple le cas du langage *Python* où l'indentation et l'espacement corrects sont des obligations fonctionnelles. En d'autres termes, si le code ne respecte pas les conventions d'indentation, celui-ci ne s'exécutera tout simplement pas.

Dans le langage PHP, le nom des fonctions joue également un aspect important dans la communication entre objets techniques. En effet, contrairement à d'autres langages de programmation (C, C++ par exemple), dans le langage PHP, un langage interprété, le code source n'est pas compilé et transformé en langage binaire. Le nom de la fonction est donc particulièrement important pour relier les pièces logicielles, au moment de l'exécution. Ces noms de fonctions sont communément appelés les API (*Application Programming Interfaces*) : lorsqu'une pièce du « puzzle » appelle une autre pièce du puzzle, elle utilise pour cela un nom normé.

> Donc puisqu'on a réalisé un ensemble de pièces de puzzle, qui fonctionnent les unes, les autres, mais qui sont codées de manières indépendantes, tu peux dans tous les cas en extraire une, et la remplacer par une pièce à toi. Pas obligé de partir avec tout le projet, de *forker* tout le projet, pour en modifier une partie.
>
> *Q- Comment les pièces se relient-elles ensemble ?*
> Elles se relient ensemble par les API, par les normes de nommage et de codage (spip03).

Ces conventions plus « fortes » sont intéressantes, car elles ont une pragmatique particulière, contrairement aux conventions faibles : elles sont régulées par la machine, par le code. Dans symfony, cette « régulation par le code » ne participe pas seulement à l'unité du code source, mais participe également à la diffusion des « bonnes pratiques ». Plusieurs des acteurs que nous avons rencontrés insistent sur la manière dont la forme « *framework* » leur permet de mieux intégrer les bonnes pratiques :

> Et à travers les *frameworks*, et ça c'est quelque chose d'important, et notamment dans symfony, c'est qu'on essaie de distiller les bonnes pratiques de développement web (sf02).
>
> Et le *framework* nous impose ses règles, quand tu dois écrire une classe, tu sais que tu dois la mettre à tel endroit dans symfony. Quand tu écris une action, tu dois la mettre à tel endroit (sf07).

On le voit ici, les conventions de nommage, en particulier lorsqu'elles sont fortes, ont une dimension prescriptive. Pour reprendre les termes de Proulx, les conventions de nommage, « à travers la forme que lui donne le concepteur, [induisent] des contraintes et une pragmatique de [leur] usage virtuel » (Proulx, 2005a, p. 313). Elles guident « l'usager » du code source dans le choix des bonnes pratiques qu'ils doit utiliser, de l'endroit où il doit placer

son code source. Dans la prochaine partie, nous nous attardons à la manière dont certaines conceptualisations du code source favorisent certaines figures d'usagers plutôt que d'autres.

5.4 Figures de l'usager et qualités du code source

Dans cette dernière partie du chapitre, nous présentons deux controverses, l'une dans SPIP et l'autre dans symfony, qui ont marqué la période de notre étude. L'idée principale que nous voulons faire ressortir de cette analyse de controverses concerne le lien entre les qualités du code source et certaines figures de l'usager (Woolgar, 1991). Ces deux controverses se rejoignent dans la mesure où elles concernent des *interfaces* à travers desquelles les acteurs peuvent agir sur le « coeur » de chacun des logiciels, et interagir entre eux. Dans symfony, la controverse concerne l'interface de programmation (API) utilisée pour générer des formulaires. Dans SPIP, elle concerne un changement proposé de la syntaxe des squelettes. Rappelons que les squelettes sont ces parties du code source qui permettent de mettre en forme les différents éléments d'un site Web (voir 4.1.3). L'analyse met donc en évidence différents choix possibles dans la conceptualisation des interfaces qui relient les différentes parties du code source, et la manière dont ces choix d'interfaces privilégient certaines figures d'usagers au détriment d'autres.

5.4.1 Étude de cas : SPIP, controverse sur une syntaxe alternative

Lors de la période de notre enquête, une importante discussion a lieu concernant le changement de syntaxe de SPIP. Nous l'avons expliqué plus tôt : construit à la base comme un système de gestion de contenu, SPIP peut être appréhendé de plus en plus comme un langage, voire même comme un cadre d'application (*framework*). Dans cette perspective, le langage de squelette, aussi appelé le *langage SPIP*, occupe une place importante. Il permet la mise en forme et la configuration d'un site ou d'une application web. Pour plusieurs acteurs, le langage SPIP est perçu comme une grande originalité du projet, en permettant à des acteurs sans connaissances avancées en programmation, de participer à la mise en forme et la configuration avancée de leur site web. Lors de la période de notre enquête, une importante discussion a lieu concernant un possible changement de syntaxe. C'est cette discussion que nous analysons ici.

La discussion en question a pour origine une conférence réalisée dans le cadre du *SPIP-Party* qui a eu lieu en juin 2009, à Avignon. Cette présentation était intitulée « *Des squelettes XML lisibles, c'est possible et c'est pour "bientôt"* ». Elle avait pour objectif principal de mettre en évidence ce qui apparaissait pour cet acteur comme des problèmes importants de la syntaxe actuelle du langage SPIP. La présentation a également mis de l'avant une proposition de syntaxe alternative pour SPIP, qui répondrait à certaines de ces critiques. La figure suivante (5.5) montre certaines différences entre la syntaxe actuelle et la syntaxe alternative proposée :

Syntaxe actuelle :

```
<B_art>
     <ul>
<BOUCLE_art(ARTICLES){id_article}>
     <li>#TITRE</li>
</BOUCLE_art>
     <ul>
</B_art>
     pas d'article
<//B_art>
```

Syntaxe alternative proposée :

```
<?spip AVANT art ?>
  <ul>
<?spip BOUCLE art ARTICLES { (id_article) ?>
  <li>#TITRE</li>
<?spip } art ?>
   <ul>
<?spip APRES art ?>
   pas d'article
<?spip VIDE art ?>
```

Figure 5.5 : Syntaxe actuelle et syntaxe alternative proposée
(À noter le code '<?spip' qui est le principal changement proposé à la syntaxe actuelle. Ce code indique un changement de langage, par exemple, entre le langage SPIP et le langage HTML)
ExtraiHt de <http://www.spip-contrib.net/Syntaxes-Alternatives-pour-SPIP> (consulté le 6 novembre 2011).

Bien que le présentateur ait insisté sur le fait que son propos visait avant tout à exposer certains problèmes plutôt qu'à proposer une syntaxe en tant que telle[144], cette proposition de syntaxe a suscité de vives réactions. La discussion s'est ensuite poursuivie dans différents espaces, et notamment sur la liste de discussion *spip-dev*. Une synthèse a également été

144 Citation : « Ce qui était important dans ma présentation, c'était l'énoncé des pbs [problèmes] qu'il me semble important de résoudre, et juste la démonstration que j'ai développé les outils pour permettre une migration en douceur de l'existant, il m'a bien fallu prendre une syntaxe nouvelle pour monter que ces outils marchaient, mais tout reste ouvert » (liste spip-dev, juin 2009).

présentée sur une page wiki[145]. La présente analyse porte plus précisément sur trois « conversations » à ce sujet, qui se sont déroulées sur la liste spip-dev[146], de même que sur les entrevues que nous avons réalisées. Cette « multi-conversation » est intéressante à analyser car elle fait ressortir les intrications entre qualités du code source et figures de l'usager.

Arguments en faveur d'un changement de syntaxe

Le principal élément soulevé pour le changement de syntaxe consiste à distinguer clairement ce qui relèverait, ou non, du langage SPIP. Un acteur, résumant cette perspective, la décrit ainsi en tentant de situer les « points d'achoppement » :

> J'ai l'impression que le point d'achoppement autour de la nouvelle syntaxe se résume à « faut-il assumer qu'un squelette mixe 2 syntaxes ou pas », c'est-à-dire que la syntaxe actuelle écrit la structure de boucle d'une manière très ressemblante à du HTML, tandis que ce que je propose est de dissocier nettement les deux. Ne pas mentir sur une réalité que tôt ou tard on finira par apprendre me paraît préférable, d'autant que les squelettes ne s'emploient pas que pour les langages de balisages (liste spip-dev, juillet 2009).

Trois points sont argumentés en faveur d'un changement de la syntaxe de SPIP. Tout d'abord, la proposition alternative (et notamment le code '<?spip') permet de rendre conforme le langage SPIP avec certains standards, comme le XML. Cet aspect contribuerait à faire reconnaître publiquement le langage SPIP comme un véritable langage de programmation, alors qu'il n'est pour l'instant pas reconnu en tant que tel à l'extérieur de la communauté de ses utilisateurs. Les fichiers écrits dans le langage SPIP pourraient ainsi porter l'extension *.spip*, alors qu'ils portent actuellement l'extension .html. Un deuxième aspect que permettrait cette conformité avec le XML et d'autres standards, est que le langage pourrait être traité par des outils standards, en particulier par des éditeurs et des analyseurs syntaxiques, qui pourraient vérifier la validité du document[147]. Finalement, un troisième argument qui est évoqué a trait à l'utilisabilité que permettrait une meilleure distinction entre les différents niveaux du langage. L'exemple d'un livre d'apprentissage d'une langue étrangère est ici donné :

---

145 <http://www.spip-contrib.net/Syntaxes-Alternatives-pour-SPIP> (consulté le 6 novembre 2011).

146 Il s'agit des conversations suivantes, sur la liste spip-dev : « Réflexion sur un changement de syntaxe des squelettes SPIP » (du 29 juin au 23 juillet 2009, 70 courriels par 17 personnes différentes); « Syntaxe alternative de spip : proposition » (du 3 septembre au 10 septembre 2009; 37 courriels par huit personnes); « Syntaxe future et premier argument de fonction » (29 octobre 2009, 5 courriels par quatre personnes).

> [Q]ue le langage produit et le langage producteur se ressemblent me paraît fondamentalement une mauvaise idée, quand bien même on ne viserait pas l'utilisation de technologies automatiques sur le langage produit. On peut faire la comparaison avec un livre d'apprentissage d'une langue étrangère : si on ne différencie pas graphiquement la langue étudiée et la langue qui expose les principes de la langue étudiée, le livre est à la limite de l'utilisable (liste spip-dev, juillet 2009).

La proposition de syntaxe alternative a cependant fait l'objet d'importantes critiques que nous présentons maintenant.

Arguments critiques du changement de syntaxe proposé

À la suite de la présentation, plusieurs courriels critiques ont été envoyés sur la liste. Il est à noter que plusieurs de ces courriels incluent des propositions précises de changements de syntaxe. Nous laissons celles-ci de côté pour nous concentrer sur les arguments et contre-arguments des acteurs. Disons simplement que la proposition officielle a reçu beaucoup de critiques pour son aspect « rigide », qui s'opposerait à une souplesse voulue qui serait nécessaire pour SPIP. Fait intéressant, en regard de notre réflexion sur la beauté, cette « souplesse » est, dans un cas, opposée à l'idée de « beauté »:

> [S]e contraindre à la conformité totale serait une erreur, car on y perdrait finalement beaucoup en souplesse au profit d'une beauté mathématique purement formelle (liste spip-dev, septembre 2009).

L'élément le plus important dans ces conversations concerne à notre avis la figure de l'usager qui est mise de l'avant. Ainsi, plusieurs arguments renvoient à l'idée d'une perte de capacité d'action du côté des « non-programmeurs » avec la syntaxe alternative (nous soulignons les passages importants en noir) :

> [U]n langage prévisible à ce point aurait toutes les chances d'être extrêmement rigide dans sa syntaxe, donc décourageant pour le **novice** et lourd à manipuler (liste spip-dev, septembre 2009).

---

147 Un aspect que nous avons complètement laissé de côté dans cette thèse, mais qui aurait pu être analysé, concerne l'usage des éditeurs de code source. Ces logiciels favorisent la lecture et l'écriture du code source, en attribuant des couleurs distinctes à chacune des composantes du code source (commentaires, noms de variable, etc). Pour cela, le format du code source doit cependant répondre à certains standards, ce qui n'est pas le cas avec la syntaxe actuelle du langage SPIP. D'où la pertinence de la syntaxe alternative, selon l'acteur qui l'a proposée.

> Plus généralement, est-il vraiment nécessaire de distinguer le passage du monde spip au monde du langage produit ? Si on se met encore une fois à la place du **néophyte**, je pense que c'est une subtilité qui risque au contraire de lui faire peur (liste spip-dev, septembre 2009).
>
> De même, [il[148]] répète souvent avec beaucoup de force qu'il ne FAUT PAS qu'il y ait de balise xml dans le code spip par peur des confusions. Je ne suis pas d'accord : ces confusions n'apparaissent qu'à **ceux qui sont suffisamment avancés en programmation** pour faire la différence (liste spip-dev, septembre 2009).
>
> Je comprends les problèmes posés par la notation des filtres postfixés (le premier argument placé avant la fonction, le deuxième après), mais elle a un énorme avantage : pour le **non programmeur**, elle affiche les appels dans leur ordre d'exécution dans le sens naturel de lecture, ce qui est un ENORME plus (liste spip-dev, septembre 2009).
>
> [P]our un **nouvel utilisateur**, c'est une écriture difficile à digérer : l'impression de rentrer dans un univers de la programmation compliquée (comme <?php) (liste spip-dev, septembre 2009).
>
> Il me semble qu'en abandonnant la syntaxe basée sur les balises <...>, on perdrait l'avantage (souvent rappelé sur spip.net) de la facilité de création / modification de squelette pour un **non programmeur** ayant simplement des bases en HTML... N'était-ce pas pourtant l'un des principaux arguments de spip ? (liste spip-dev, juillet 2009).

À l'inverse, un autre acteur propose quant à lui d'abandonner le langage SPIP de façon à favoriser une autre figure d'usager : le « développeurs PHP ».

> Je me pose aussi la question : serait-il pertinent de permettre également l'écriture de squelettes avec une syntaxe purement php ? Spip se comporterait alors simplement comme une API[149], et serait plus facilement accessible aux **développeurs php** « durs » qui n'auraient pas à apprendre une nouvelle syntaxe (liste spip-dev, juillet 2009).

Ces différents extraits sont particulièrement importants en regard de notre étude car ils montrent que les choix entourant le design « interne » du code source entraînent, pour employer les termes de Woolgar (1991), une certaine « configuration de l'usager ». Rappelons que pour Woolgar, la définition d'une machine s'élabore mutuellement avec celle de son contexte (voir chapitre 2, p. 68). Selon Woolgar, cette élaboration mutuelle est

---

148 Nom enlevé pour préserver l'anonymat.

149 API : Application Programming Interface.

particulièrement marquée pour cet aspect du contexte appelé l'usager[150] : « The capacity and boundedness of the machine take their sense and meaning from the capacity and boundedness of the user » (Woolgar, 1991, p. 68). Les descriptions précédentes permettent de faire ressortir cet aspect de définition mutuelle de la machine et de l'usager : les choix de conception des différentes parties du code source sont étroitement liés à l'idée qui est faite de l'utilisateur de cette partie du code source.

Sur le plan empirique, insistons encore sur ce discours, très présent dans SPIP, qui veut qu'un langage informatique n'est pas uniquement destiné à « ceux qui sont suffisamment avancés en programmation ». Dans la perspective de ces acteurs, du moins, il est possible de créer un langage informatique, et par le fait même, un code source, qui serait accessible pour des « novices », « néophytes » et autres « non-programmeurs ». Ces propos montrent bien que le code source n'est pas, du moins selon les acteurs de SPIP, un artefact dont l'usage serait par essence interdit aux non programmeurs. Au contraire, la capacité d'action de l'utilisateur apparaît intimement liée à la manière dont les différentes interfaces qui composent le code source – les squelettes dans ce cas-ci – sont conçues[151].

5.4.2 Étude de cas : symfony, la controverse sur les formulaires et la complexification du *Framework*

Dans le cadre de symfony, la controverse analysée a trait au format par défaut pour configurer les formulaires. La controverse s'est en fait déroulée un peu avant le début de notre enquête de terrain, mais était néanmoins très présente durant nos entrevues et nos observations. En effet, juste avant le début de notre enquête, l'un des acteurs fortement impliqués dans le projet avait décidé de quitter l'équipe de *core*, après avoir exprimé publiquement son désaccord avec certaines orientations du projet et ce, sur plusieurs listes de discussions et blogues publics. Ce départ a d'ailleurs créé une sorte de commotion dans la communauté. Lors de la première conférence de symfony à laquelle nous avons participé, plusieurs des participants avaient

150 « this part of context called the user » (Woolgar, 1991, p. 68).

151 C'est d'ailleurs dans cette perspective que nous proposons d'appréhender en conclusion de la thèse le code source en tant qu'interface des reconfigurations humain-machine. Si le code source est une interface à travers laquelle les acteurs peuvent agir sur la machine, symétriquement, le design du code source – en tant qu'interface – privilégie certains types d'usages et certaines catégories d'usagers plutôt que d'autres.

d'ailleurs soulevé des inquiétudes concernant l'impact de ce départ sur la communauté. L'année suivante, l'acteur démissionnaire a toutefois décidé de participer à nouveau dans symfony, mais dans un rôle plus périphérique.

Cette controverse est intéressante à présenter ici pour plusieurs raisons. D'une part, bien qu'elle se soit déroulée avant le début de notre enquête, cette controverse est significative des principaux problèmes vécus dans symfony. Un des acteurs de symfony nous indique par exemple que, dans symfony, « l'essentiel des problèmes est lié aux formulaires » (sf08). La controverse est également intéressante, car, tout comme dans SPIP, elle met en jeu différentes manières de juger de la qualité du code. Finalement, comme dans SPIP, la controverse a trait à un aspect d'*interface* qui permet à des usagers externes au *core* d'interagir avec le code source, en le configurant[152]. Dans le cadre de cette section, nous reconstituons la controverse en nous appuyant principalement sur nos entrevues, mais également sur l'analyse de certaines discussions et certains billets de blogues.

La controverse en question concerne une nouvelle façon de configurer et gérer les formulaires web, qui a été introduite dans la version 1.1 de symfony. Un « formulaire web » désigne dans symfony toute combinaison de champs d'entrées de données sur une page web, par exemple les deux champs utilisateur/mot de passe pour s'inscrire à un site web; ou encore un champ de commentaire. Dans symfony, donc, tous les champs d'entrées de données sur le web sont considérés sous l'appellation de formulaires. Dans symfony 1.0, les formulaires étaient principalement gérés à l'aide du langage de programmation nommé YAML (*YAML Ain't Markup Language*). Dans une version subséquente (la version 1.1), l'utilisation du langage YAML a été abandonnée, et l'emploi direct du langage PHP préconisé en utilisant une approche orientée-objet, qui s'appuyait sur le *design pattern* des Modèle-Vue-Controlleur (MVC), que nous avons présentés plus tôt. Un acteur du projet (qui critique cette approche), décrit ce qui, selon lui, a conduit à ce choix :

---

152 Mentionnons également que, contrairement à la controverse que nous avons analysée dans SPIP, la controverse sur les formulaires dans symfony semble renvoyer à des aspects plus émotifs et des conflits de personnalité entre les acteurs impliqués. Nous évitons cependant d'aborder cette dimension du conflit, pour nous concentrer ici sur des aspects qui nous semblent cruciaux liés à la configuration des interfaces du code source.

> C'est-à-dire qu'il s'est dit, « le MVC est un motif de conception logicielle reconnu, qui a fait ses preuves. Symfony lui-même est totalement basé sur cette architecture en trois tiers » et il s'est dit « je vais transposer ce pattern sur la gestion des formulaires » (sf08).

La figure 5.6, à la page suivante, montre les deux manières de faire, pour construire un formulaire similaire.

Cette manière de conceptualiser la gestion des formulaires a suscité plusieurs réactions. En effet, selon notre interlocuteur (sf08) que nous avons cité plus tôt, « l'essentiel des problèmes est lié aux formulaires » (sf08), et ce, malgré le fait que trois versions successives de symfony aient été publiées. Il nous explique par exemple qu'il est extrêmement difficile, dans cette nouvelle architecture des formulaires, de simplement afficher un astérisque pour marquer le caractère obligatoire d'un champ de formulaire, sans contourner complètement l'approche choisie. L'extrait suivant de l'entrevue explique ce problème, en mettant de l'avant une figure de l'usager, celle de l'*intégrateur* :

> C'est-à-dire que le premier besoin d'un, d'un **intégrateur**, par exemple, c'est d'afficher le caractère requis d'un champ. En HTML, on met habituellement un petit astérisque à côté d'un champ requis. Et ça, c'était pas possible et ça ne l'est toujours pas, à part en contournant totalement le principe d'architecture qui a été implémenté dans le *framework* de formulaire, donc ça s'appelle... un *hack* (sf08).

Deux critiques sont faites à l'endroit de la nouvelle approche de gestion des formulaires basée sur PHP, le modèle MVC. Selon cet acteur, il y a d'abord un problème de conception au niveau de l'architecture : « il y avait clairement un gros problème de cohérence sur l'application d'un motif de conception vis-à-vis de l'objectif-cible d'exploitation » (sf08). Ainsi, si cet acteur est d'accord avec le choix d'utiliser PHP plutôt que YAML, le recours au modèle MVC lui semble toutefois problématique[153].

153 Mentionnons par ailleurs que pour cet acteur, la publication du « *framework* de formulaires » a été réalisée trop rapidement, rapidité qui était due à des contraintes liées à des engagements commerciaux – des « urgences *business* » (sf08) – car le réalisateur du projet s'était engagé auprès de partenaires à publier la version stable la semaine suivante. L'un des principaux problèmes liés à cette nouvelle implantation de la gestion des formulaires était donc, simplement, que ceux-ci n'étaient pas prêts au moment de la sortie.

Configuration en format PHP (nouveau format) :

```
<?php
class ContactForm extends sfForm
{
  protected static $subjects = array('Subject A', 'Subject B', 'Subject C');

  public function configure()
  {
    $this->widgetSchema->setNameFormat('contact[%s]');
    $this->widgetSchema->setIdFormat('my_form_%s');
    $this->setWidgets(array(
      'name'    => new sfWidgetFormInput(),
      'email'   => new sfWidgetFormInput(),
      'subject' => new sfWidgetFormSelect(array('choices' => self::$subjects)),
      'message' => new sfWidgetFormTextarea(),
      'file'    => new sfWidgetFormInputFile(),
    ));

    $this->setValidators(array(
      'name'    => new sfValidatorString(array('required' => false)),
      'email'   => new sfValidatorEmail(),
        'subject' => new sfValidatorChoice(array('choices' => array_keys(self::
$subjects))),
              'message'  =>  new  sfValidatorString(array('min_length'  =>  4),
array('min_length' => 'Your message is too short')),
      'file'    => new sfValidatorFile(),
    ));
    $this->setDefault('email', 'me@example.com');
  }
}
?>
```

Configuration en format YAML (ancien format) :

```
# in YAML
&subjects:     [Subject A, Subject B, Subject C]
name_format:  contact[%s]
id_format:    my_form_%s
widgets:
  name:       text
  email:      { type: text, default: me@example.com }
  subject:    { type: select, choices: *subjects }
  message:    textarea
  file:       file
validators:
  name:       { type: string, required: false }
  email:      email
  subject:    { type: choice, choices: *subjects }
  message:    { type: string, min_length: 4, errors: { min_length: Your message
is too short } }
  file:       file
```

Figure 5.6 : Différents formats pour la configuration des formulaires
En haut : la configuration des formulaires en PHP. En bas, la configuration des formulaires en YAML.
Source : <http://redotheweb.com/category/usability> (consulté le 8 novembre 2011).

L'autre critique concerne cependant des questions d'ergonomie et de lisibilité, qui sont plus proches de notre intérêt de recherche. L'extrait suivant d'une entrevue décrit bien les différentes positions dans la controverse, entre le choix d'utiliser YAML, et celui d'utiliser PHP (nous soulignons les passages importants) :

> L'affrontement que B a eu avec G portait plus sur, euh, sur l'**ergonomie du code** [...]. C'est-à-dire qu'il voulait qu'on puisse configurer les formulaires et les procédures dans un langage qui s'appelle le « YAML » [prononcé « yammeul »], qui ressemble un peu à un XML, mais dans **une syntaxe beaucoup plus lisible**. Et l'argumentation de G vis-à-vis de cette proposition a toujours été de dire, « Je préfère quand même **que les gens écrivent du PHP**, c'est-à-dire le langage qu'exploite avec le *framework*, pour décrire **les choses complexes** (sf08).

D'autres extraits d'entrevues permettent d'explorer ces positions des acteurs. Ainsi, cet acteur, favorable au maintien du format YAML pour configurer les formulaires, explique que les changements apportés ont pour conséquence de nuire à la facilité d'utilisation :

> Il m'a semblé que symfony au départ répondait à ce besoin de simplicité à destination des développeurs. [...] Pour les versions suivantes, on a voulu prendre un peu plus de hauteur et restructurer ça de manière un peu différente. Pour ça, on a fait des changements qui selon moi... nuisaient à la facilité d'utilisation de certains points (sf03).

Plus intéressant pour notre analyse, cet acteur mobilise cette fois la catégorie de la beauté en la posant en contradiction avec celle de la facilité d'utilisation :

> Je me mets à la place du développeur et je me dis, ah, c'est du beau code, quand on le voit. Mais quand on commence à l'utiliser, finalement, c'est pas très pratique! Ils auraient pu le faire un peu moins beau, mais plus facile à utiliser ! C'est des contradictions qui deviennent fondamentales quand elles sont érigées en principe (sf03).

Un autre acteur, favorable au nouveau système de formulaire, met de l'avant les catégories de sécurité et d'extensibilité du code source, pour justifier sa complexité :

> Un truc qu'on a eu pas mal de controverses, c'est le nouveau système de formulaires qu'on a introduit dans symfony 1.1, qui était beaucoup plus compliqué que ce qu'on avait avant, puisqu'avant, on n'avait rien donc euh. C'était compliqué, mais qui avait, qui a le gros avantage de structurer énormément les choses, d'assurer la sécurité, d'assurer l'extensibilité. Et, je suis persuadé que c'est une bonne chose, et certainement que plein de gens qui pensaient que c'est une mauvaise chose parce que ça devenait trop compliqué (sf02).

Cet acteur, qui remarque que cette complexité accrue du *framework* est l'une des sources principales de tension dans symfony, justifie toutefois cette complexité par le fait que le *framework* n'est pas destiné pour le monde :

> Donc, les tensions, on les a parce qu'il y a des gens qui trouvent que symfony est beaucoup trop compliqué. Et j'ai envie de dire c'est pas grave. Ça veut dire simplement dire que le *Framework* n'est pas fait pour ces gens-là (sf02).

Ce dernier extrait d'entrevue montre d'ailleurs bien comment une certaine complexification du *framework* peut avoir comme conséquence assumée d'exclure certains types d'usagers. Plus précisément, la figure de l'usager qui semble être exclue par cette complexification du *framework* est celle du « bidouilleur », de ces gens dont « c'est pas forcément le métier, mais qui utilise[nt] PHP parce que c'est très simple à apprendre » (sf02). Le bidouilleur semble être exclu au profit du professionnel, « des gens qui utilisent PHP en entreprise » (sf02). Cet extrait d'entrevue est très clair à ce propos :

> Ce qui fait que la cible de symfony ou du Zend framework, typiquement ces deux là, ou EasyComponents, est une cible très restreinte par rapport à la communauté globale de PHP. C'est une cible plutôt professionnelle, plutôt de gens dont c'est le métier. Donc plutôt de gens qui ont un bagage technique assez important (sf02).

Cette présentation des différentes positions dans la controverse sur les formulaires et la complexification du *framework* montre bien les liens étroits entre choix d'écriture ou de design, qualités du code source et certaines figures de l'usager. Comme c'était le cas dans SPIP, la controverse sur les formulaires dans symfony montre comment la capacité d'action de certaines catégories d'usagers peut être renforcée ou réduite par les choix de design du code source. Plus fondamentalement, ces controverses mettent de l'avant l'idée que le code source n'est pas, par essence, une boîte noire inaccessible aux utilisateurs « non-programmeurs », mais que l'accessibilité est plutôt négociée et construite par les acteurs qui décident de la forme que ce code source prendra. C'est dans cette perspective que nous proposons, en conclusion de la thèse, d'aborder le code source comme interface, ou comme multiplicité d'interfaces dans les reconfigurations humain-machine, et dont la conception est négociée et entraîne nécessairement certains ordonnancements et/ou exclusions.

5.5 Conclusion partielle : « mon interface, c'est le code »

> Quand je suis un graphiste, mon interface, c'est Photoshop, avec ses boutons, ses fenêtres, etc. Quand je suis un développeur, mon interface, c'est le code. C'est avec ça que j'interagis avec la matière que je construis, étant un programme. Alors que le graphiste il va construire une image (sf03).

L'analyse des qualités du code source permet de faire ressortir le caractère collectif de sa fabrication ainsi que sa proximité avec une forme d'écriture. Cette écriture collective peut s'apparenter à l'écriture d'un ouvrage d'une certaine envergure, par exemple la présente thèse ou encore un ouvrage collectif regroupant plusieurs articles. On retrouverait probablement dans la rédaction d'un tel ouvrage des critères de propreté, voire d'élégance, qui vont dans le sens d'une uniformisation et d'un respect des styles d'écriture. La particularité de l'écriture du code source – si l'on se limite à sa dimension écrite[154] – est que celle-ci est cependant très fortement distribuée. L'écriture du code source est une écriture à plusieurs mains. Plusieurs acteurs peuvent écrire chacun un morceau de code source qui sera ensuite assemblé à un « corpus » plus large dans une dynamique similaire au « copier-coller » décrit par cet acteur :

> Il faut bien voir que la façon de coder, enfin c'est aussi ma façon de coder, je ne tape pas beaucoup de code. C'est énormément de copier-coller. De récupération de bouts de code, parce que la structure, les fautes de frappe et tout, on en enlève pas mal quand on récupère des bouts entiers (sf04).

Dans cette perspective, le code source constitue, on l'a vu, une forme d'écriture particulièrement normée. Ainsi, le « style d'écriture » auquel fait référence Knuth (1974) lorsqu'il insiste sur l'importance d'adopter un style qui permet de bien s'exprimer, s'apparente plus à la notion de style mobilisée dans les expressions telles que les « feuilles de style » ou le « style de formatage » que l'on retrouve dans les logiciels de traitement de texte, ou bien dans la fabrication de sites web. De plus, le choix d'un « style » d'écriture particulier, ainsi que le respect des conventions de nommages renvoient à une problématique communicationnelle. Ainsi, la qualité de lisibilité n'a rien à voir avec le fonctionnement du programme et l'exécution de la machine, mais est plutôt mise de l'avant pour faciliter la collaboration entre les acteurs. De même, les conventions d'écriture, en particulier lorsqu'elles

---

154 Comme nous l'avons noté au chapitre précédent, bien que le code source ne soit pas toujours un écrit, il prend généralement cette forme, ce qui rend à notre avis pertinent le recours à la métaphore de l'écriture pour décrire la fabrication collective du code source.

sont « faibles », ont surtout pour rôle de faciliter le partage du code source et la collaboration entre les acteurs.

D'une certaine manière, les qualités du code source et les convention de nommage participent également à la performativité du code source. Comme le notent Denis et Pontille (2010a), la performativité des artefacts repose en effet – du moins dans la perspective de la théorie de l'acteur-réseau – sur la stabilité qu'ils entretiennent dans un réseau. Citant Latour (1985) Denis et Pontille notent que « la force des artefacts est conditionnée à leur capacité à devenir des "mobiles immuables", c'est-à-dire des objets qui peuvent circuler d'un point à un autre du réseau sans changer d'état, ni perdre leur forme » (Denis et Pontille, 2010a, p. 111). Les normes d'écriture, en particulier dans le contexte du logiciel libre, participent en quelque sorte à cette « immuabilité mobile ». Alors que les licences de logiciels libres permettent le libre partage du code source, les normes d'écriture et les conventions facilitent l'insertion réussie d'un morceau de code source dans un réseau ou une « chaîne de code source » élargie. Ainsi, pour qu'un morceau de code source gagne en force, il doit être inséré dans un ensemble plus grand de code source qui comprend des règles et des normes d'écriture. L'« immuabilité mobile » du code source apparaît d'autant plus lorsqu'il s'agit de faire circuler un morceau de code source d'un réseau à un autre : plus les normes et conventions d'écritures entre ces entités sont communes, plus le morceau de code source peut « circuler » (être reproduit en fait) d'un réseau à l'autre.

Les normes et les conventions d'écriture participent également d'une certaine configuration de l'usager et d'une inscription des valeurs de chacun des projets au sein du code source. De manière générale, nous avons mentionné que SPIP semble rejoindre un public plus homogène en termes d'origine nationale – surtout des Français –, mais plus diversifié en termes de compétences, en regroupant par exemple des acteurs qui n'ont à peu près pas d'expérience en informatique[155]. Symfony pour sa part, vise un public international, mais peut-être moins

155 Nous pourrions argumenter que le choix de cette interface, en permettant aux « novices » de participer, est une bonne chose politiquement. Toutefois, certains participants de SPIP font remarquer que les « clients » qui utilisaient SPIP auparavant se tournent désormais vers WordPress où l'interface est encore plus facile d'accès, en partie parce qu'elle est dénudée de code. D'une certaine façon, l'analyse du « langage SPIP » montre bien la manière dont le code source agit comme une interface dans les assemblages humain-machine, mais, d'une autre manière, force est de constater que d'autres interfaces peuvent faire le même travail, sans nécessairement relever de code source.

diversifié en termes de compétences, surtout des informaticiens professionnels. Dans symfony, les choix d'écriture reposent donc davantage sur des « bonnes pratiques » et des standards reconnus que dans le cas de SPIP. Le fait que le code source soit écrit en anglais, et que toutes les communications de la communauté se fassent dans cette langue, constitue également une certaine forme d'inscription des valeurs de la communauté dans le code source du projet. Nous avons également noté une différence importante entre SPIP et symfony au niveau de la mise en place des normes d'écriture et de la rigueur avec laquelle celles-ci sont suivies. Dans symfony, les normes d'écriture sont beaucoup plus rigoureuses et ont été établies dès le début du projet. Dans SPIP, au contraire, le projet a débuté sans nécessairement avoir de normes d'écriture et, bien que certaines règles aient été mises en place, le code source de SPIP reste en général assez faiblement uniformisé. Cette distinction exprime encore une fois une certaine inscription des valeurs dans le code source de chacun des projets. Les valeurs des projets s'expriment également dans le choix de l'anglais ou du français comme langue du code source. Elles sont également particulièrement notables dans notre analyse des débats dans chacun des projets concernant les choix de la syntaxe pour le langage de squelette dans SPIP ou du format utilisé pour la configuration des formulaires dans symfony (section 5.1). Cette analyse mettait de l'avant la manière dont certaines parties du code source agissent comme des interfaces à travers desquelles les acteurs peuvent en quelque sorte étendre, ré-assembler, re-configurer d'autres parties du code source. Plus important encore, le design de ces interfaces implique, comme pour celui de toute interface, certains découpages qui excluront éventuellement de certaines figures de l'usager au profit d'autres.

Le code source n'est pas seulement le produit de la conception technologique, mais il participe, en lui-même, à la collaboration entre les acteurs. Dans ce sens, les propos de l'acteur que nous avons cité en exergue rejoignent un aspect essentiel du code source que nous voulons mettre de l'avant dans cette thèse. Pour cet acteur, le code source agit à la manière d'une interface, au même titre que les boutons ou les fenêtres dans d'autres logiciels graphiques. C'est particulièrement le cas pour ces aspects du code source, comme les fichiers ou les formats de configuration présentés à la section 5.4, qui permettent d'« interagir » avec d'autres réseaux de code source, placés dans une boîte noire. Toutefois, le code source agit

également comme interface dans la collaboration des humains entre eux. C'est cette conception étendue de l'interface qui alimentera la conclusion de notre thèse, au chapitre 7.

CHAPITRE VI

LE COMMIT COMME ACTE « AUTORISÉ » DE CONTRIBUTION AU CODE SOURCE

> Je commite donc je suis ! Il y a un coté très symptomatique de l'époque à laquelle on vit !
>
> - Un acteur de SPIP[156]

Ce chapitre prend comme point de départ l'analyse de l'acte informatique du commit. Le « commit » est une commande informatique implantée dans plusieurs logiciels de gestion des versions du code source. Dans les projets étudiés, le commit devient verbe et l'acte de commiter constitue souvent l'étape ultime pour valider une modification au code source. Le fil rouge de notre analyse consiste à appréhender le commit à la manière d'un acte qui participe à la validation du code source « autorisé » à figurer sur la page de téléchargement et qui couronne le travail d'une contribution au code source. Nos descriptions empiriques s'appuient particulièrement sur l'analyse des droits et des autorisations entourant l'acte du commit, en insistant sur les manières dont ces autorisations sont intimement liées aux valeurs des projets étudiés et entraînent différentes dynamiques de visibilité. Après avoir introduit le concept de « code source autorisé » (partie 6.1), nous présentons deux études de cas, l'une pour SPIP et l'autre pour symfony, afin d'explorer le travail de négociation qui s'articule autour de l'acte du commit (partie 6.2). Nous décrivons ensuite plus en détails les différentes autorisations auxquelles est articulé l'acte du commit (partie 6.3), puis faisons ressortir les dynamiques de visibilité et d'autorité autour de cet acte, que nous appréhendons également comme un acte de contribution (partie 6.4).

6.1 Le code source « autorisé »

Dans le chapitre 4, nous avons présenté différentes formes et statuts du code source en insistant notamment sur la manière dont pour certains acteurs – en particulier dans symfony –

---

156 Extrait d'une entrevue réalisée avec un acteur de SPIP. Comme nous l'écrirons aussi plus loin, nous préférons ne pas donner ici l'identifiant de l'entrevue en question, afin de préserver l'anonymat de cet acteur dans les autres propos présentés dans la thèse.

le code source du projet est « celui que je peux télécharger » (sf05) ou encore que « quand on télécharge symfony, on télécharge les sources de symfony » (sf07). Nous avons également mentionné que ce code source, disponible sur la page de téléchargement, constitue uniquement cette partie du code source qui a été décrétée à un moment donné, et par certaines personnes, comme étant une version *stable*. Nous proposons ici de considérer ce code source comme étant le code source « autorisé », en ce sens d'abord trivial qu'il s'agit du code source *autorisé* à figurer sur la page de téléchargement du projet. Le caractère « autorisé » que nous mettons ici de l'avant renvoie également à certaines analyses de la performativité. Dans le monde francophone, la plus connue de celles-ci est sans doute la critique que fait Bourdieu (1975) à Austin (1975), où il développe le concept de « langage autorisé » pour insister sur les conditions sociales de la performativité du langage. Pour Bourdieu, un énoncé est performatif non pas uniquement parce qu'il obéit à certaines conventions, mais parce que son énonciateur est institutionnellement autorisé à le prononcer. Joerges (1999), citant notamment ce texte de Bourdieu, mobilise également le concept d'autorisation pour analyser le pouvoir des artefacts. Critiquant le populaire essai de Winner (1980), « Do Artifacts Have Politics? » (1980), l'auteur soutient que le pouvoir des dispositifs techniques ou construits (*built*) ne doit pas être trouvé dans des attributs formels des choses elles-mêmes, mais plutôt dans leur *autorisation*, ce qu'il définit comme leur « représentation légitimée » (*legitimate representation*), qui donne une forme aux effets concrets des choses (Joerges, 1999, p. 429)[157]. Dans son étude sur la performativité du code informatique, Mackenzie (2005) met également de l'avant la notion similaire d'*authorizing context*. Cette notion, que nous pourrions traduire par le terme de « *contexte d'autorisation*[158]», serait définie par l'ensemble des conventions et des pratiques dont la répétition donne sa force à l'acte performatif[159].

---

157 Joerges critique ici le texte de Winner (1980) qui considérait que les ponts de la ville de New York, conçus par Robert Moses dans les années 1920, « faisaient de la politique » dans ce sens qu'ils empêchaient alors les pauvres, et en particulier les Noirs, d'accéder aux plages new-yorkaises. En effet, ces ponts étaient conçus tellement bas qu'ils ne pouvaient pas permettre le passage des autobus, empêchant ainsi leurs utilisateurs – surtout des pauvres – d'accéder à certaines zones de New York, contrairement aux automobiles et leurs propriétaires. Joerges constate toutefois que bien que ces ponts existent toujours, ils n'ont plus aujourd'hui cette capacité politique discriminatoire. C'est dans ce sens que Joerges considère que les artefacts trouvent leur capacité politique non pas en eux-mêmes, mais plutôt dans les droits et la culture qui « légitiment » ou « autorisent » cette capacité politique.

Dans le cadre de la présente analyse, nous voulons cependant faire référence aux travaux d'anthropologie de l'écriture pour aborder ces « conditions de félicité » qui donnent au code source une « autorisation » d'agir. À la suite de Bourdieu (1975), notre concept de « code source autorisé » permet de mettre de l'avant le fait qu'il ne s'agit pas seulement d'écrire un morceau de code source pour que celui-ci agisse, il faut encore que celui-ci soit *autorisé* à agir. Contrairement à l'analyse de Bourdieu, toutefois, les autorisations entourant le code source ne renvoient pas seulement à la position sociale de la personne qui écrit le code, mais plutôt à l'insertion de ce morceau de code à l'intérieur du code source qui est autorisé à figurer sur la page de téléchargement. Ce morceau de code doit pour cela être articulé à un réseau sociotechnique élargi, qui comprend des ensembles plus larges de réseaux de code source, mais également des dispositifs techniques d'*autorisation*, de même que certains signes de validation (le premier étant la page de téléchargement). Cette perspective rejoint l'analyse que Fraenkel fait de la performativité des actes d'écrits. À propos du testament, elle note ainsi que la performativité de cet artefact dépend en bonne partie de son insertion dans « un système de chaînes d'écriture, de personnes habilitées et de signes de validation, éléments qui forment l'authenticité nécessaire à la performativité » (Fraenkel, 2006)[160].

L'apposition d'un certain nombre de signes (sceaux, tampons, signatures) sont des actes qui, en donnant son authenticité au document légal, participent à sa performativité. C'est donc à une anthropologie pragmatique de l'écriture, qui s'attache à étudier des actes courants d'écriture, tels que la signature ou l'étiquetage, que Fraenkel nous convie (Fraenkel, 2006).

---

158 Il est par ailleurs intéressant de noter ici que les termes d'auteur, d'autorisation et d'autorité ont la même racine latine soit « fondement ». À cet effet, nous souhaitons renvoyer à l'excellent essai de Hannah Arendt « Qu'est-ce que l'autorité ?» (Arendt, 1972) qui nous a également beaucoup inspirés, de façon implicite, dans notre réflexion sur l'autorité. À des fins de cohérence avec notre cadre théorique, nous préférons cependant ne pas aborder cette auteure dans le corps du texte.

159 « The "authorizing context" is the set of conventions and practices whose repetition gives a performative act its force » (Mackenzie, 2005, p. 82).

160 Mentionnons ici que la comparaison du code source avec les actes écrits de Fraenkel est limitée. En effet, comme nous l'avons noté au chapitre 4, le code source ne se restreint pas à son caractère écrit. Il conviendrait donc de généraliser cette perspective pour appréhender le code source dans la perspective d'une réseau sociotechnique complexe, plutôt que simplement une chaîne d'écriture. Il semble que le recours à l'analyse de Fraenkel a l'avantage de bien faire ressortir cette dimension écrite du code source qui, bien que limitée, reste néanmoins prégnante.

C'est dans cette perspective que nous proposons d'aborder l'acte du commit en insistant ensuite sur les différents enjeux d'autorité et de visibilité qui participent à cet acte[161].

6.2 Le commit, acte « autorisé » d'écriture du code source

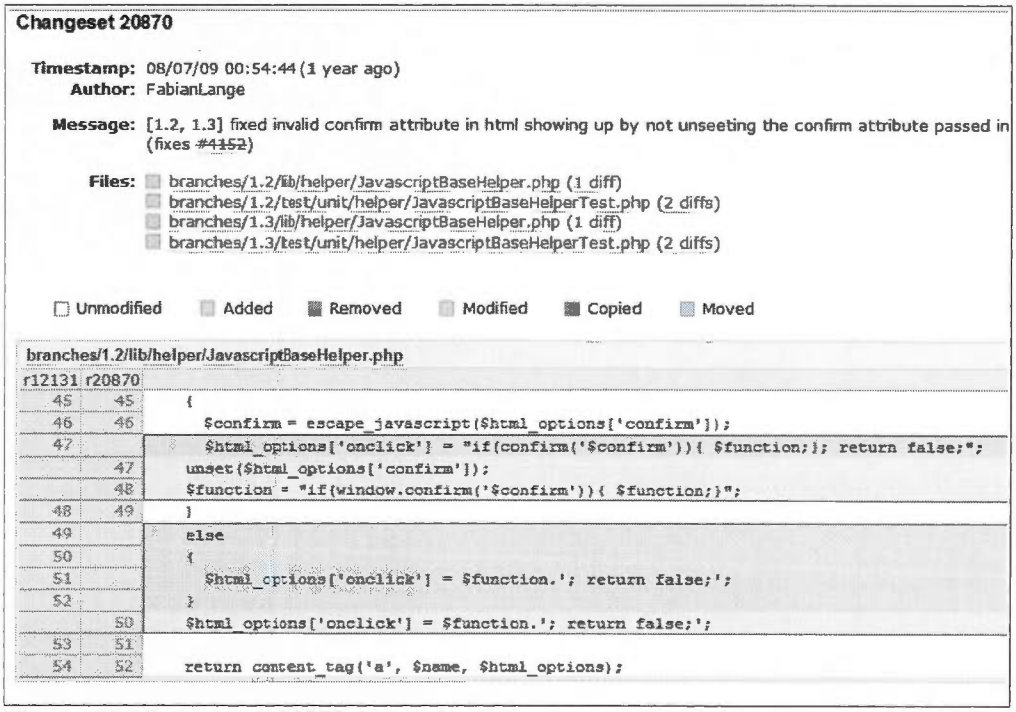
Plusieurs actes participent à la performativité du code source, ou à son « autorisation ». Nous nous intéressons dans ce chapitre au commit, un acte similaire à la signature dans la théorie des actes d'écriture de Fraenkel, qui participe à la performativité du code source, à son *autorisation*. Le commit est une commande informatique implantée dans plusieurs systèmes de gestion de versions, et qui consiste à valider une modification au code source. SPIP et symfony, comme de nombreux projets de développement de logiciels (libres), s'appuient sur des dispositifs facilitant la gestion des différentes versions du code source. Dans les projets étudiés, le logiciel en question est *Subversion*, logiciel également connu par l'acronyme SVN. Bien qu'utilisé principalement dans le monde de l'informatique pour gérer le code source, le logiciel Subversion peut en fait gérer tout type de document en format texte. Il est par exemple utilisé dans symfony pour faciliter l'écriture de la documentation technique. La transaction de base dans ce logiciel est la commande informatique *commit*. Le commit consiste à publier les changements apportés au code source, pour les rendre accessibles aux autres acteurs impliqués dans le projet, voire même, dans le cas de logiciels libres, au public en général[162]. Ainsi, un acteur peut travailler sur le code source du logiciel sur son ordinateur

---

161 Il nous a été mentionné que le commit pourrait être comparé à d'autres actes de langage, tels que l'acte « commissif » dans la classification de Searle (1976), qui renvoie à un acte où le locuteur s'engage à réaliser une action future (par exemple : « je serai à l'heure du rendez-vous »). Outre le fait que nous n'avons pas du tout exploré les travaux de Searle dans notre cadre théorique, il nous semble que les ressemblances entre le commit et l'acte « commissif » se limitent au terme (commettre). En effet, comme nous le verrons plus loin, l'acte du commit, malgré son nom, n'est aucunement lié à une action future. Son action est plutôt immédiate et consiste à valider une modification au code source dit « autorisé ». C'est dans cette perspective que le commit, en tant qu'acte de validation, nous semble plus près de l'acte de signature analysé par Fraenkel (2006; 2008) : le commit est en quelque sorte par rapport au code source, ce que la signature est par rapport au testament. L'important dans le présent chapitre n'est pas tant de restituer la spécificité de l'acte du commit vis-à-vis d'autres actes de langage et d'écriture, mais plutôt de « suivre » cet acte pour analyser les dynamiques complexes de transformation du code source, en tentant de faire ressortir le rôle de cet acte quant à la performativité du code source (ou d'un morceau de code source).

162 La charte de fonctionnement de SPIP explicite le terme commiter par le fait d'« ajouter ou modifier ("commiter") des fichiers dans le répertoire Subversion (SVN) ».
<http://zone.spip.org/trac/spip-zone/wiki/CharteDeFonctionnement> (consulté le 3 octobre 2011).

personnel, en version locale, puis rendre publics ses changements à l'aide de la commande commit. Chacun des commits est associé, dans le système de gestion de versions, à un numéro de révision, au nom de l'utilisateur qui a publié le commit, de même qu'à un message optionnel (appelé *log*, en anglais) et l'heure à laquelle a été réalisé le commit (le *timestamp*). Le commit décrit également chacun des fichiers du code source qui a été modifié. Ces informations sont ensuite accessibles à travers différentes interfaces et permettent de conserver l'historique des modifications apportées au code source. Plusieurs modifications peuvent donc être réalisées plusieurs fois par jour, occasionnant chacune une nouvelle révision du code source. La figure suivante (6.1) présente une mise en forme des données associées à un commit dans symfony. Elle montre également la manière dont les modifications au code source sont faites petit à petit, voire ligne par ligne, dans le temps.



Changeset 20870

Timestamp: 08/07/09 00:54:44 (1 year ago)
Author: FabianLange

Message: [1.2, 1.3] fixed invalid confirm attribute in html showing up by not unseeting the confirm attribute passed in (fixes #4152)

Files:
branches/1.2/lib/helper/JavascriptBaseHelper.php (1 diff)
branches/1.2/test/unit/helper/JavascriptBaseHelperTest.php (2 diffs)
branches/1.3/lib/helper/JavascriptBaseHelper.php (1 diff)
branches/1.3/test/unit/helper/JavascriptBaseHelperTest.php (2 diffs)

Unmodified Added Removed Modified Copied Moved

branches/1.2/lib/helper/JavascriptBaseHelper.php

r12131	r20870	
45	45	{
46	46	$confirm = escape_javascript($html_options['confirm']);
47		$html_options['onclick'] = "if(confirm('$confirm')){ $function;}; return false;";
	47	unset($html_options['confirm']);
	48	$function = "if(window.confirm('$confirm')){ $function;}";
48	49	}
49		else
50		{
51		$html_options['onclick'] = $function.'; return false;';
52		}
	50	$html_options['onclick'] = $function.'; return false;';
53	51	
54	52	return content_tag('a', $name, $html_options);

Figure 6.1 : Le commit 20870 de symfony
<http://trac.symfony-project.org/changeset/20870> (consulté le 11 novembre 2011).

L'acte d'étiquetage, pratiqué à l'aide du logiciel Subversion, est également important à mentionner ici. Cet acte consiste à étiqueter une révision spécifique du code source pour

décréter qu'il s'agit, par exemple, de la version « stable » disponible en téléchargement. D'une certaine manière, cet acte possède une pragmatique plus forte que celle du commit, puisqu'il s'agit de l'acte qui autorise effectivement une révision particulière du code source à figurer sur la page de téléchargement. L'effet du commit se limite plutôt à autoriser une modification au code source dans le dépôt officiel du projet. Nous avons toutefois choisi de consacrer notre analyse au commit, étant donné l'importance que prend cet acte dans les projets de logiciels libres (ce que nous montrons plus loin).

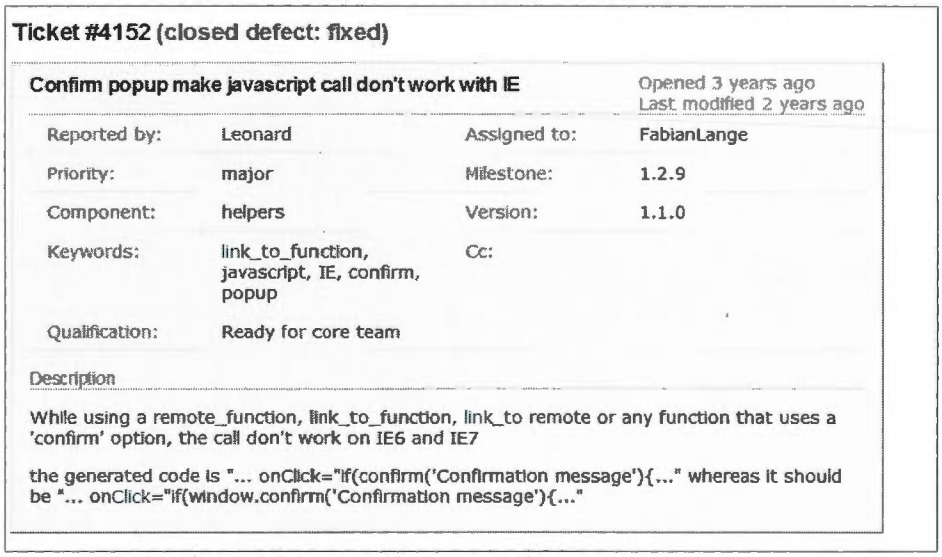
Plusieurs modifications peuvent donc être réalisées plusieurs fois par jour, occasionnant chacune une nouvelle révision du code source. Or, comme l'écrit Fraenkel à propos de la signature, celle-ci « se présente comme un acte pris dans une action plus large. Elle suppose plusieurs actants, plusieurs auteurs, plusieurs mains » (Fraenkel, 2008). Qui plus est, dans le cadre des projets étudiés, l'acte du commit est régi par différents droits et autorisations qui marquent, comme les études mentionnées plus tôt le notent, certains rapports d'autorité entre les membres. Dans les deux prochaines sous-sections, nous étudions le cas de deux commits réalisés dans chacun des projets, en insistant sur le travail nécessaire à l'accomplissement de ces commits.

6.2.1 Étude de cas : le commit 20870 de symfony

Nous décrivons ici le cas du commit 20870 du logiciel symfony. Ce commit a été réalisé dans le code source du « *core* » de symfony[163]. Notre description s'appuie sur l'analyse du système *trac*, que nous décrivons un peu plus loin. Comme il sera décrit plus loin, l'acte du commit est régi par différents droits et autorisations et seules certaines personnes peuvent commiter sur une partie donnée du code source. Plusieurs pratiques et dispositifs sont cependant mis en place pour assurer une participation plus étendue. L'une de ces pratiques consiste à soumettre des rustines (des « *patches* » en anglais) et des rapports de bogues, qui peuvent être suivis dans le temps. Ces possibilités de contributions se font par l'intermédiaire d'un logiciel nommé *trac* qui permet de suivre ces propositions de modifications au code source, autour

163 Comme décrit dans le chapitre 4, autant dans SPIP que dans symfony, le « *core* » constitue le noyau du logiciel. Il est parfois appelé le « coeur » par les acteurs, en particulier dans SPIP. Le « *core* » est à distinguer des « *plugins* » qui sont des modules, ou des extensions, qui peuvent venir s'ajouter au logiciel.

d'un *ticket*[164]. Un ticket est une requête, ouverte par un usager concernant une défectuosité à corriger ou une fonctionnalité à ajouter, et qui peut ensuite faire l'objet de différents commentaires ou interventions de la part des acteurs impliqués. Certaines méta-informations sont associées au ticket en question, par exemple, si son statut est « ouvert » ou « fermé », s'il a été résolu, s'il concerne un problème « majeur » ou « mineur », etc. Le ticket peut également contenir des propositions concrètes de modifications au code source de la part des acteurs. La figure suivante (6.2) présente une capture d'écran du *ticket* identifié par le numéro 4152 sur lequel notre analyse s'appuie :



Ticket #4152 (closed defect: fixed)

Confirm popup make javascript call don't work with IE

Opened 3 years ago
Last modified 2 years ago

Reported by:	Leonard	Assigned to:	FabianLange
Priority:	major	Milestone:	1.2.9
Component:	helpers	Version:	1.1.0
Keywords:	link_to_function, javascript, IE, confirm, popup	Cc:	
Qualification:	Ready for core team		

Description

While using a remote_function, link_to_function, link_to remote or any function that uses a 'confirm' option, the call don't work on IE6 and IE7

the generated code is "... onClick="if(confirm('Confirmation message'){..." whereas it should be "... onClick="if(window.confirm('Confirmation message'){..."

Figure 6.2 : Entête du ticket no 4152 de symfony
<http://trac.symfony-project.org/ticket/4152> (consulté le 8 octobre 2011).

Nous avons choisi d'analyser un ticket pour le cas de symfony puisque le système de tickets fait l'objet d'un usage important. Pour trouver le ticket en question, nous avons restreint notre choix aux tickets qui avaient été effectivement fermés par l'équipe de coeur (le *core team).* Parmi ces différents tickets, nous en avons choisi un qui permet d'analyser un commit qui mobilise plusieurs acteurs, dispositifs techniques, ainsi que plusieurs interventions dans le

164 Le logiciel *trac* contient également une interface pour visualiser les informations gérées par SVN (le code source, de même que les « logs »), de même qu'un wiki, pour réaliser de la documentation collaborative.

temps. Notre choix s'est finalement arrêté au ticket numéro 4152[165]. Ce ticket a été ouvert le 7 août 2008 à 17h26 et a suscité 17 interventions s'échelonnant sur une période d'un an, soit jusqu'au 7 août 2009. Sept personnes ont participé aux discussions concernant ce ticket. Parmi celles-ci, deux étaient membres du *core team* et avaient par conséquent les droits de commit nécessaires pour publier des modifications au code source de symfony[166]. Sans entrer dans les détails du contenu de la modification proposée (détails que nous avons nous-mêmes de la difficulté à comprendre), disons que ce ticket concerne un bogue lié à la génération automatique de code Javascript dans le navigateur Internet Explorer. La première intervention est celle de Leonard, qui a ouvert le ticket et a ensuite proposé une série de modifications qu'il décrit comme un « *patch* » (une rustine, en français), spécifiant les lignes du code source qui doivent être modifiées (figure 6.3).

Figure 6.3 : Proposition d'une rustine pour symfony
<http://trac.symfony-project.org/ticket/4152> (consulté le 8 octobre 2011).

165 <http://trac.symfony-project.org/ticket/4152> (consulté le 8 octobre 2011).

166 Le ticket en question étant clairement accessible publiquement, et la présente analyse ne comportant à notre avis aucun risque pour les personnes citées, nous avons décidé, à des fins de clarté, de conserver tels quels les pseudonymes utilisés par les acteurs (nous les enlevons cependant plus loin dans le texte, lorsque ces références sont trop explicites).

En même temps que son intervention, Leonard modifie les champs « statut » et « résolution » de son ticket à respectivement « closed » et « fixed », ce qui signifie que le ticket est fermé. Trois heures plus tard, une seconde personne intervient pour obtenir une clarification concernant le statut du ticket : « Is this fixed? or is just ready for review? ». Ceci amène une réponse, deux jours plus tard par l'auteur du premier message (Leonard) qui spécifie avoir proposé une solution (« a fix ») pour aider d'autres personnes de la communauté à résoudre le problème. Encore deux journées plus tard, un membre de l'équipe de coeur intervient, en spécifiant que selon lui, la proposition amenée plus tôt apparaît adéquate et pourrait être appliquée à la version 1.x.

Aucune modification au ticket n'est cependant apportée pendant plusieurs mois, jusqu'à ce qu'un autre acteur, Andromeda, s'interroge sur le fait que cette solution (« this fix ») n'a pas encore été appliquée à la version 1.2.5, alors que celle-ci pourrait être d'une grande aide. Un peu moins d'une heure plus tard, on retrouve six interventions sur une période de deux minutes de la part d'un membre du *core team*, disposant des droits d'écriture pour modifier le code source de symfony. Ces interventions spécifient que les commits portant les numéros 17383, 17384, 17385 et 17386 ont été réalisés, un pour chacune des branches correspondant respectivement aux versions 1.3, 1.2, 1.1 et 1.0 du code source de symfony. Cet intervenant modifie également les méta-informations concernant ce ticket, entre autres pour indiquer que le statut du ticket est désormais fermé (« closed ») (figure 6.4).

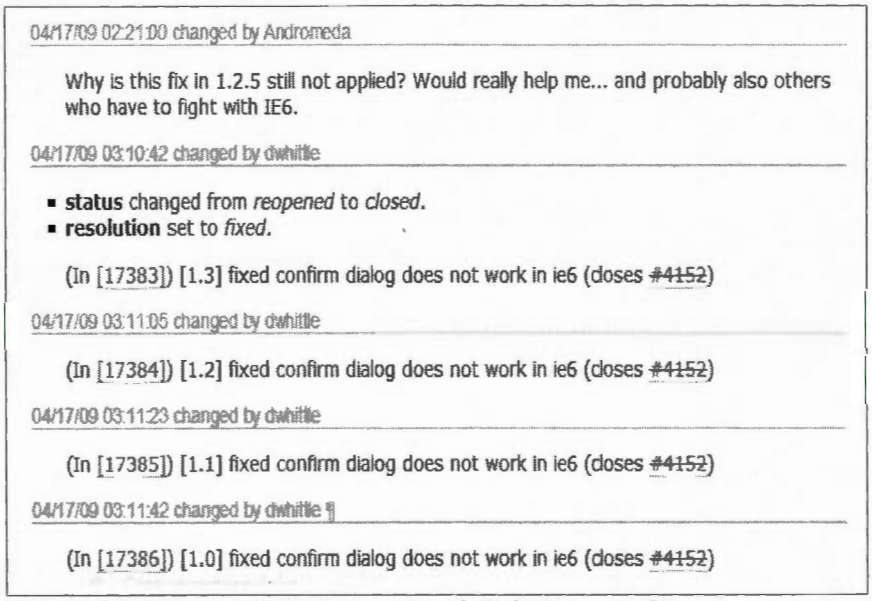


Figure 6.4 : Intervention et commits réalisés dans symfony
<http://trac.symfony-project.org/ticket/4152> (consulté le 17 novembre 2011).

La conversation reste ensuite silencieuse pendant plus de deux mois, jusqu'au 28 juin 2009, au moment où un quatrième acteur, Intru, entre en scène. Celui-ci décide d'ouvrir de nouveau le ticket, en demandant si la correction ne devrait pas être appliquée ailleurs (figure 6.5).

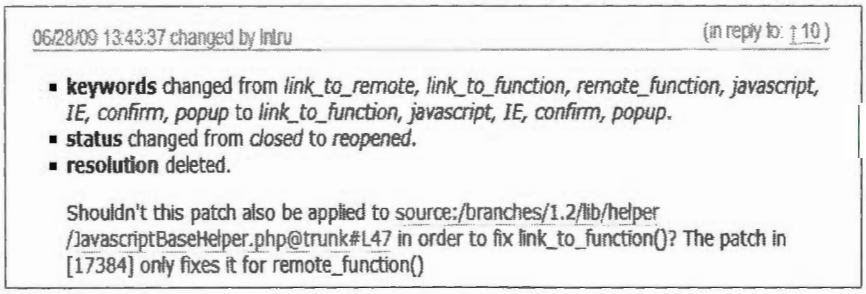
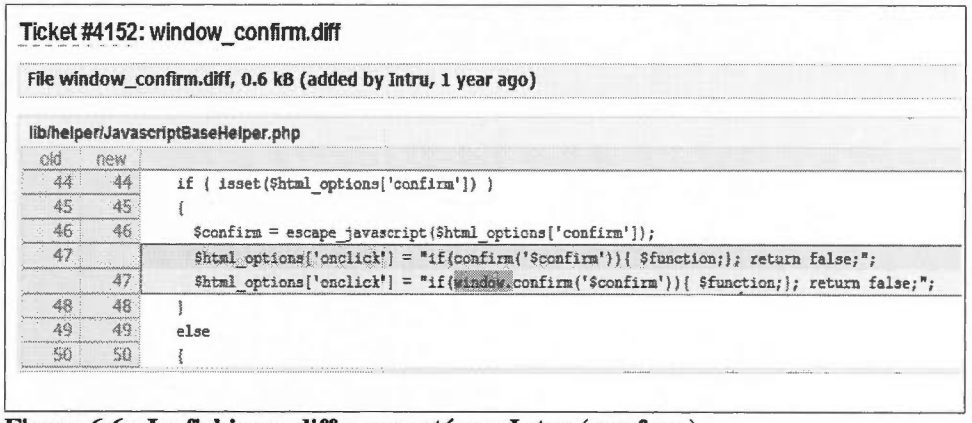


Figure 6.5 : Réouverture du ticket par Intru (symfony)
<http://trac.symfony-project.org/ticket/4152> (consulté le 17 novembre 2011).

Douze jours se font encore attendre, puis le même Intru décide de soumettre en pièce jointe un fichier dont l'extension est .diff. Ce fichier, dont on retrouve une mise en forme à la figure suivante (6.6), présente ligne par ligne, les modifications proposées. Contrairement à la proposition de rustine de la figure 6.3, le fichier .diff constitue une rustine qui peut être appliquée de façon plus automatisée par des personnes disposant des droits d'écriture.



Ticket #4152: window_confirm.diff

File window_confirm.diff, 0.6 kB (added by Intru, 1 year ago)

lib/helper/JavascriptBaseHelper.php

old	new	
44	44	`if ( isset($html_options['confirm']) )`
45	45	`{`
46	46	`  $confirm = escape_javascript($html_options['confirm']);`
47		`  $html_options['onclick'] = "if(confirm('$confirm')){ $function;}; return false;";`
	47	`  $html_options['onclick'] = "if(window.confirm('$confirm')){ $function;}; return false;";`
48	48	`}`
49	49	`else`
50	50	`{`

Figure 6.6 : Le fichier « .diff » apporté par Intru (symfony)
<http://trac.symfony-project.org/ticket/4152> (consulté le 17 novembre 2011).

Après presque un mois de silence, un sixième acteur, Boutell, entre en scène le 6 août 2009, en indiquant que la proposition présentée plus tôt n'est pas idéale et constitue simplement un pansement temporaire. Cet acteur propose une autre solution (« a fix »), plus complexe. Le lendemain, un septième acteur, FabienLange, qui est membre de l'équipe de coeur, et qui possède donc les droits d'écriture conséquents, décide de commiter ce nouveau code, et de fermer le ticket. Ceci met fin à la saga du ticket 4152. Ces changements sont connus sous le nom de la révision 20870, dont nous avons présenté une capture d'écran du commit à la figure 6.1. Notons par ailleurs que les informations contenues dans le commit montrent que le « propriétaire » du billet a été réassigné de fabien, qui est le chef du projet symfony, à FabienLange, le membre du *core team* qui a effectivement fermé le ticket.

6.2.2 Étude de cas : le commit 37817 de SPIP

Dans le cas de SPIP, notre analyse s'attarde aux commits 37817 et 37868 dans la *Zone* de SPIP, c'est-à-dire dans l'espace où est déposé le code source des plugins. Nous avons choisi de décrire cette suite de commits pour porter l'attention sur les conséquences pragmatiques de l'acte du commit et montrer comment, dans SPIP, cet acte entraîne parfois l'ouverture d'une discussion par courriel, pouvant éventuellement conduire à l'annulation du commit.

Le point de départ de notre description concerne le commit 37817, dont on peut voir une capture d'écran à la figure suivante (6.7) :

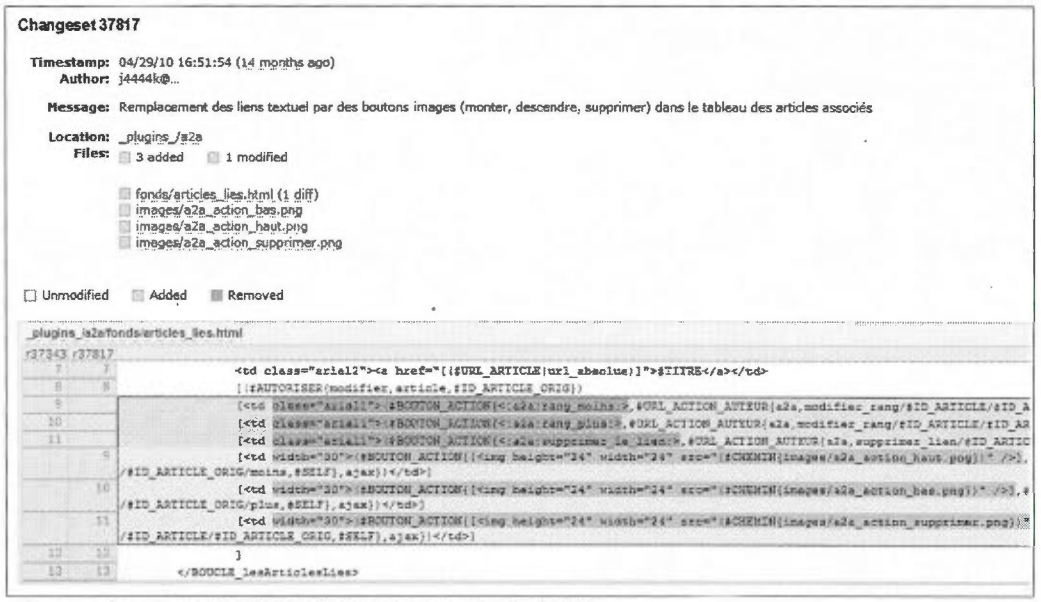


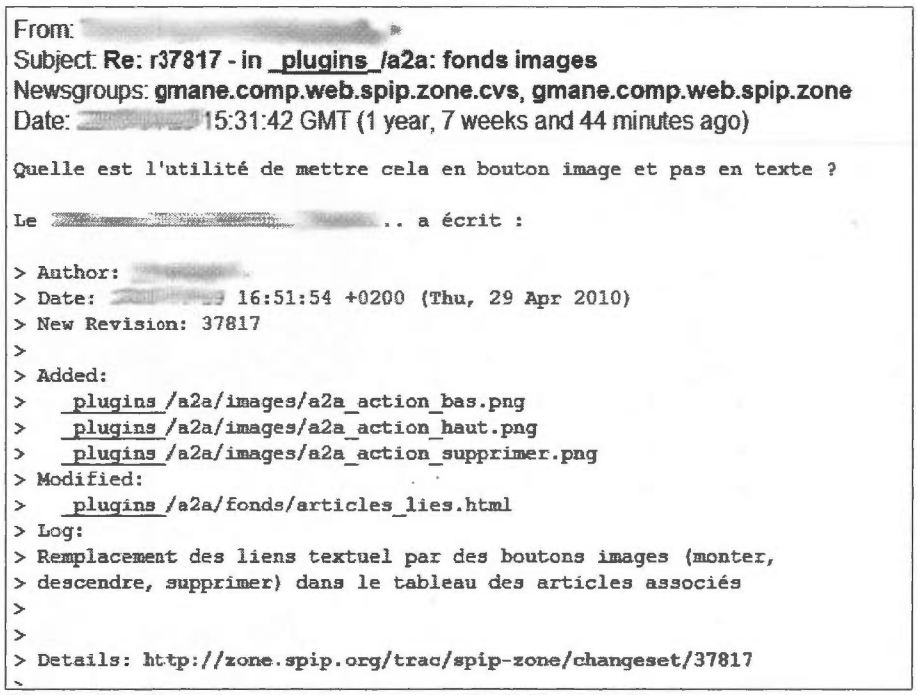
Figure 6.7 : Le commit 37817 sur la zone de SPIP
<http://zone.spip.org/trac/spip-zone/changeset/37817> (consulté le 11 novembre 2011).

Inutile de comprendre ici en détails les implications de cette modification à l'interface du logiciel. Tenons-nous en au *log* : il s'agit d'une modification du code source du plugin *a2a* qui consiste à remplacer des liens textuels par des boutons images. L'un des aspects intéressants de ce commit est qu'il ne concerne pas uniquement du texte. Trois images – des fichiers de format .png – sont ajoutées au code source du plugin. En effet, à la figure précédente (6.7), on retrouve le nom de ces images dans la section spécifiant les images ajoutées, de même que dans les lignes 9, 10 et 11 du code source modifié. Cette situation montre bien, comme nous l'avons soutenu au chapitre 4, la manière dont le code source ne peut pas être uniquement appréhendé comme un écrit, ou comme du texte. Au contraire, le texte qui forme le code source est également étroitement intriqué à d'autres médias dont il est difficile de statuer s'ils font partie ou non du code source.

Le commit 37817, comme tous les commits dans la *Zone*, a par la suite déclenché l'envoi automatique d'un courriel sur la liste [spip-zone-commit]. Notons en effet que plusieurs opérations peuvent être associées au commit. Ainsi, dans le cas de SPIP, le commit déclenche automatiquement l'envoi d'un courriel à différentes listes de courriel, soit les listes [spip-

commit] dont le descriptif est « SPIP : toutes les modifs du code[167] » et [spip-zone-commit] dont le descriptif est « SPIP zone est un espace de développement collaboratif pour SPIP[168] ». À chaque commit effectué, un courriel est envoyé sur l'une des deux listes[169].

Le courriel en question a donc été envoyé sur la liste [spip-zone commit] à 16h51. À 17h31, soit 40 minutes plus tard, un participant au projet répond au courriel en questionnant l'utilité de mettre le bouton en image. Il met également en copie conforme la liste [spip-zone]. Nous retrouvons une capture d'écran de ce courriel à la figure 6.8.



From: [redacted]
Subject: **Re: r37817 - in _plugins_/a2a: fonds images**
Newsgroups: **gmane.comp.web.spip.zone.cvs, gmane.comp.web.spip.zone**
Date: [redacted] 15:31:42 GMT (1 year, 7 weeks and 44 minutes ago)

```
Quelle est l'utilité de mettre cela en bouton image et pas en texte ?

Le [redacted] .. a écrit :

> Author: [redacted]
> Date: [redacted] 16:51:54 +0200 (Thu, 29 Apr 2010)
> New Revision: 37817
>
> Added:
>    plugins/a2a/images/a2a_action_bas.png
>    plugins/a2a/images/a2a_action_haut.png
>    plugins/a2a/images/a2a_action_supprimer.png
> Modified:
>    plugins/a2a/fonds/articles_lies.html
> Log:
> Remplacement des liens textuel par des boutons images (monter,
> descendre, supprimer) dans le tableau des articles associés
>
>
> Details: http://zone.spip.org/trac/spip-zone/changeset/37817
```

Figure 6.8 : Première réaction au commit 37817 (SPIP)
<http://thread.gmane.org/gmane.comp.web.spip.zone.cvs/33122/focus=17738>
(consulté le 9 novembre 2011).

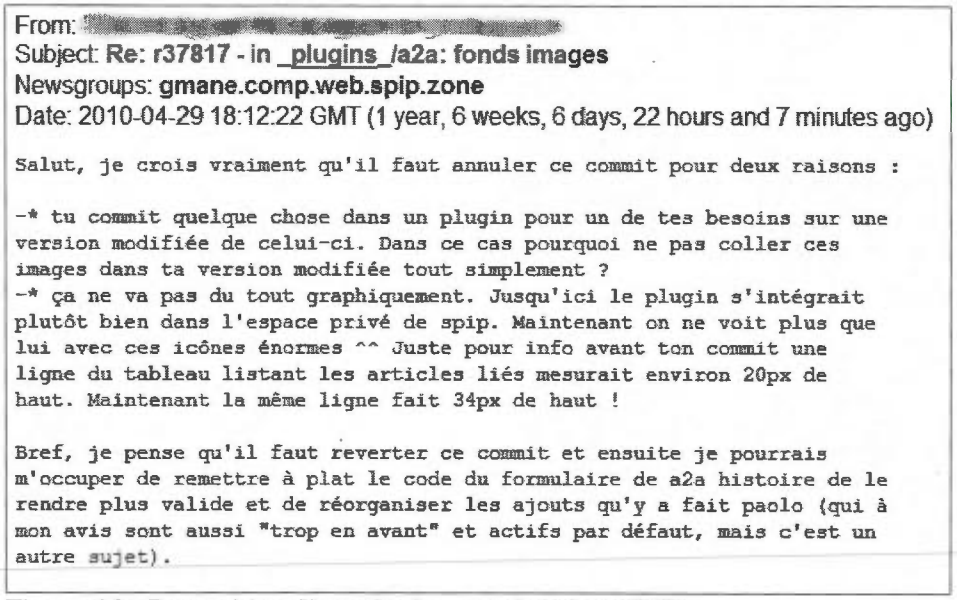
---

167 <http://listes.rezo.net/mailman/listinfo/spip-commit> (consulté le 9 novembre 2011).

168 <http://listes.rezo.net/mailman/listinfo/spip-zone-commit> (consulté le 9 novembre 2011).

169 Le commit peut également être associé à des fils RSS qui peuvent ensuite être articulés à d'autres sites web ou à des lecteurs de nouvelles, dans une logique d'agrégation typique des sites « web 2.0 » et décrite notamment par Blondeau (2007). Ces derniers aspects, les courriels et les fils RSS, montrent bien comment les morceaux de code source peuvent circuler à la manière d'autres écrits du web 2.0, tels les billets de blogues, ou les entrées de Twitter. Nous reviendrons sur cet aspect dans la présentation de Git et GitHub plus loin dans ce chapitre.

À la suite de ce courriel, un second acteur questionne à son tour le commit qui vient d'être fait. Le commiteur répond ensuite aux questions soulevées pour justifier son commit. En réponse à ce courriel, un quatrième acteur propose alors explicitement, vers 20h12 le même jour, d'« annuler ce commit », en expliquant ses raisons. Il propose lui-même d'apporter de nouvelles améliorations (figure 6.9).



From: [redacted]
Subject: **Re: r37817 - in _plugins_/a2a: fonds images**
Newsgroups: **gmane.comp.web.spip.zone**
Date: 2010-04-29 18:12:22 GMT (1 year, 6 weeks, 6 days, 22 hours and 7 minutes ago)

```
Salut, je crois vraiment qu'il faut annuler ce commit pour deux raisons :

-* tu commit quelque chose dans un plugin pour un de tes besoins sur une
version modifiée de celui-ci. Dans ce cas pourquoi ne pas coller ces
images dans ta version modifiée tout simplement ?
-* ça ne va pas du tout graphiquement. Jusqu'ici le plugin s'intégrait
plutôt bien dans l'espace privé de spip. Maintenant on ne voit plus que
lui avec ces icônes énormes ^^ Juste pour info avant ton commit une
ligne du tableau listant les articles liés mesurait environ 20px de
haut. Maintenant la même ligne fait 34px de haut !

Bref, je pense qu'il faut reverter ce commit et ensuite je pourrais
m'occuper de remettre à plat le code du formulaire de a2a histoire de le
rendre plus valide et de réorganiser les ajouts qu'y a fait paolo (qui à
mon avis sont aussi "trop en avant" et actifs par défaut, mais c'est un
autre sujet).
```

Figure 6.9 : Proposition d'annuler le commit 37817 (SPIP)
<http://thread.gmane.org/gmane.comp.web.spip.zone.cvs/33122/focus=17738>
(consulté le 9 novembre 2011).

Notons dans ce courriel le passage faisant état d'un code source personnalisé : « Dans ce cas pourquoi ne pas coller ces images dans ta version modifiée tout simplement? ». Selon cet intervenant, un des problèmes du commit en question est qu'il n'a pas un intérêt général et que le changement proposé devrait donc se limiter à une version personnalisée. Ce passage permet de comprendre un peu mieux les différents statuts du code source : d'une part, un code source « personnalisé », se situant sur la machine personnelle; d'autre part, un code source « public » qui peut être modifié par l'intermédiaire du commit. Un troisième niveau de code source constituerait le code source *autorisé*, soit celui qui est étiqueté comme stable à un moment donné.

Le lendemain, l'acteur qui a le premier commité annonce qu'il a décidé d'annuler le commit en réalisant le commit 37868, une sorte de contre-commit, qui a pour message « Suppression du commit... » (figure 6.10).

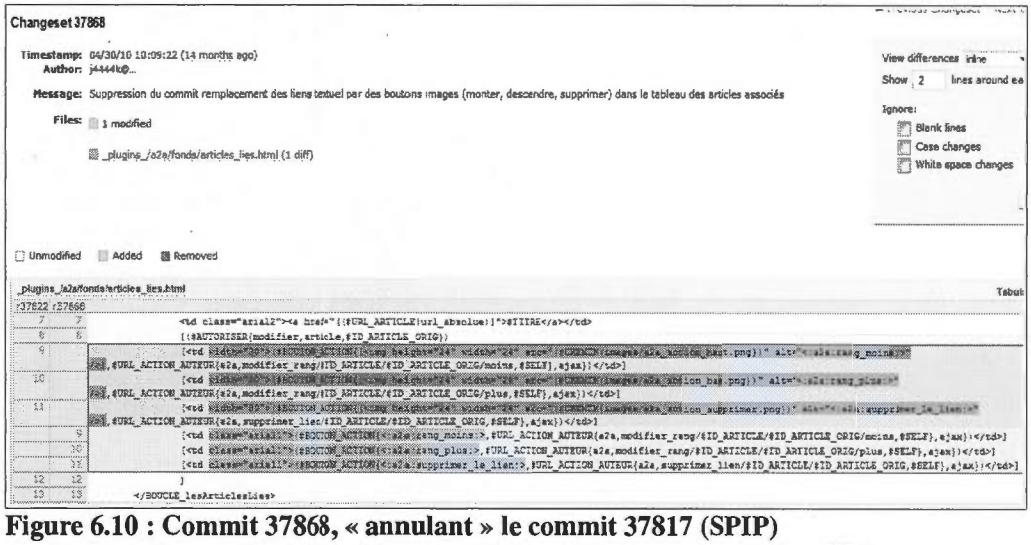


Figure 6.10 : Commit 37868, « annulant » le commit 37817 (SPIP)
<http://zone.spip.org/trac/spip-zone/changeset/37868> (consulté le 11 novembre 2011).

Deux éléments d'analyse ressortent de cette description. Tout d'abord, on voit bien la manière dont les modifications au code source sont parfois validées a posteriori. Comme indiqué précédemment, dans la *Zone* de SPIP, c'est-à-dire dans le dépôt du code source des plugins et des squelettes, les autorisations d'écriture sont accordées à tout le monde, sous condition d'adhérer à la charte de SPIP ainsi qu'à certaines règles présentées plus loin. Cette situation fait en sorte que quiconque possède les droits d'écriture peut commiter partout sur la *Zone*. La discussion se fait donc a posteriori et l'autorité se situe ailleurs que dans les dispositifs techniques, mais vraisemblablement, dans l'autorité morale des acteurs de la *SPIP Team* qui ont, dans ce cas-ci, insisté pour que le commit soit annulé.

Notons par ailleurs que le mode de validation a posteriori semble également être en vigueur en ce qui concerne le code source du *core*, bien qu'un nombre moins important d'acteurs aient le droit de commiter. Certaines tensions existent en effet entre les acteurs du *core*, ce qui fait qu'on retrouve souvent l'épisode du commit/dé-commit :

> Ah bien ça, ça s'est produit plein de fois. Il y en a un qui commite un truc, et puis l'autre qui dit : « moi je ne suis pas d'accord, je fais un *revert* ». Il y a l'autre qui peut s'amuser. L'autre va faire sa petite vengeance. Chacun est libre de commiter et de dé-commiter. Mais bon, après ça devient puéril, tu vois (spip08).

Dans ce cas, l'autorité, au final, se situe davantage dans l'autorité morale de l'un des fondateurs du projet :

> Clairement, quand il dit quelque chose, de toute façon, ça fait, et bien c'est [il] a parlé tu vois. Même si ce n'est pas dit, même si c'est le créateur de SPIP. C'est celui qui a toujours ouvert les clés à tout le monde. Il te donne les clés très très facilement. Mais en même temps, quand ça dérape, etc., et bien clairement, c'est lui qui dit « voilà ». Et ça fait foi, quoi (spip08).

6.3 Les droits de commiter dans les projets étudiés

Les descriptions précédentes ont permis de contraster différentes manières de valider et d'autoriser des modifications au code source. Dans le premier cas (le commit 20870 de symfony), le commit constitue en quelque sorte le couronnement d'un travail de négociation qui se déroule sur une période d'une année. Dans le second cas (le commit 37817 sur spip-zone), au contraire, le commit est plutôt le point de départ d'une discussion qui conduit finalement à un second commit visant à « dé-commiter » le premier. Ce fil de l'action distinct pour chacun des deux projets s'explique notamment par des manières également distinctes de mettre en œuvre les autorisations de commiter, que les acteurs désignent comme des « droits de commiter » (spip01) ou encore des « droits d'écriture » (sf07). Seules certaines personnes possèdent effectivement le droit de commiter et l'octroi de ces droits est par ailleurs intimement lié à l'organisation de la communauté et à la hiérarchisation de ses membres. Plusieurs pratiques et dispositifs sont cependant mis en place pour assurer une participation plus étendue. Ces différentes façons de faire rendent également compte de différentes manières d'articuler les droits de commiter dans chacun des projets. Dans cette partie, nous présentons les différentes manières dont sont gérés ces droits pour chacun des projets étudiés.

6.3.1 Droits de commiter dans le « *core* » de symfony et de SPIP

L'une des démarcations importantes dans chacun des projets est sans doute celle entre les acteurs qui possèdent le droit de commiter dans le *core* du code source, et les autres qui ne possèdent pas ces droits. Dans chacun des projets, le nombre de personnes qui possèdent ces

droits est réduit à une dizaine, tout au plus. Cette limitation des droits en écriture, bien que nécessaire pour assurer une certaine stabilité au code source, est cependant problématique, car elle a pour conséquence de limiter les possibilités de participation. Toutefois, comme dans SPIP, on retrouve dans symfony plusieurs dispositifs qui sont mis en place pour assurer une participation étendue.

Un premier moyen consiste à permettre aux acteurs de soumettre des propositions de modification sous forme de rustine (*patch*), en particulier par l'utilisation du système *trac*. L'étude de cas du commit 20820 de symfony, présenté à la section 6.2.1, a permis de décrire cet usage de la rustine et du système trac. L'utilisation du système de tickets est particulièrement importante au sein de symfony, et a même donné lieu, en 2010, à un concours au sein de cette communauté. L'événement aux allures festives, intitulé 1day1ticket[170], consistait pour les participants à fermer un billet par jour, par exemple, en envoyant une rustine au code source du projet. Dans SPIP toutefois, l'utilisation du système de tickets est beaucoup moins importante, probablement parce que les droits en écriture sont accordés de façon beaucoup plus libérale : le fait d'accorder à un plus grand nombre de personnes des droits en écriture permet évidemment la modification directe du code, plutôt que de passer par une rustine.

L'autre moyen principal utilisé pour assurer une participation étendue malgré ces limitations des droits de commiter est de permettre la production des plugins. Comme décrit dans le chapitre 4, les plugins sont en quelque sorte des modules extensibles et relativement indépendants qui peuvent être développés par les contributeurs du projet qui ne font pas partie du coeur. Le recours à une architecture de plugins constitue en quelque sorte un moyen pour décentraliser la participation à la fabrication du code source. Ainsi, dans les deux projets, on retrouve la création d'un répertoire de plugins dont l'accès « en écriture » est régi par des autorisations distinctes de celle du *core*. Le développement de plugins constitue donc une évolution commune à chacun des deux projets. Toutefois, la gestion précise des droits en écriture sur les plugins s'opère distinctement dans le cas de SPIP et de symfony, différences qui reflètent des distinctions en termes de compétences techniques, mais également en termes de culture politique.

---

170 <http://trac.symfony-project.org/wiki/1day1ticket> (consulté le 9 novembre 2011).

6.3.2 Les plugins de symfony

> Aujourd'hui, typiquement, j'ai vu qu'il y avait un bug quelque part, j'ai corrigé chez moi en version locale. Et par contre, je ne pouvais pas le commiter. Ça marchait chez moi, les modifications, elles sont faites sur ma machine. Du coup, j'ai vu qu'il y avait un bug, j'ai envoyé un mail sur la *mailing list* pour savoir comment il fallait faire. Donc, là il m'a demandé de m'inscrire en tant que développeur sur le plugin. Et là, j'ai le droit d'écriture dessus (sf07).

Dans symfony, la gestion des droits d'écriture dans les plugins se fait plugin par plugin. Comme indiqué précédemment, dans le cas de symfony, les plugins et le « *core* » se situent dans le même dépôt de gestion du code source. Dans ce dépôt, cependant, les droits en écriture sur chacun des répertoires sont gérés de façon distincte. Chacun des plugins possède ses propres droits en écriture et certaines personnes possèdent une autorité pour accorder un droit en écriture à d'autres contributeurs des plugins.

Pour contribuer à un plugin, il faut d'abord s'inscrire et se connecter sur le site des plugins de symfony (http://www.symfony-project.org/plugins). Il faut ensuite aller dans l'onglet « Contribuer » du plugin en question auquel on désire contribuer. La prochaine figure montre cet onglet (figure 6.11) :

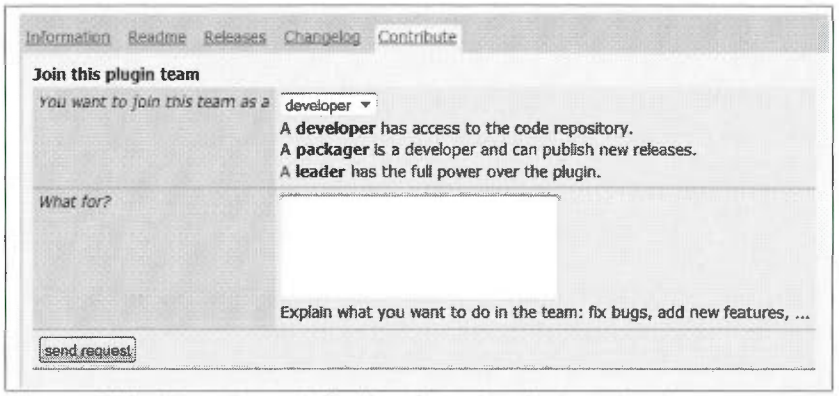


Figure 6.11 : Page de contribution à un plugin (symfony)
<http://www.symfony-project.org/plugins/sfSympalPlugin> (consulté le 9 novembre 2011).

L'onglet « Contribuer » est intéressant car il montre différents niveaux d'autorisation dans la gestion du code source des plugins. Le « développeur » a accès au dépôt du code source du

plugin. Le « *packager* » est celui (ou celle) qui peut publier des nouveaux « *releases* » et le « leader » a tous les pouvoirs autour du plugin (c'est-à-dire qu'il peut modifier le code, publier de nouvelles versions et accorder ces droits à d'autres personnes). Comme indiqué précédemment, chaque plugin est géré de façon indépendante et constitue donc en soi un projet autonome. Une personne qui possède tous les droits sur un plugin n'a donc pas nécessairement de droits sur un autre plugin.

Un autre niveau d'autorisation est celui lié à la création de plugins, dont la figure suivante (6.12) montre une capture d'écran[171] :

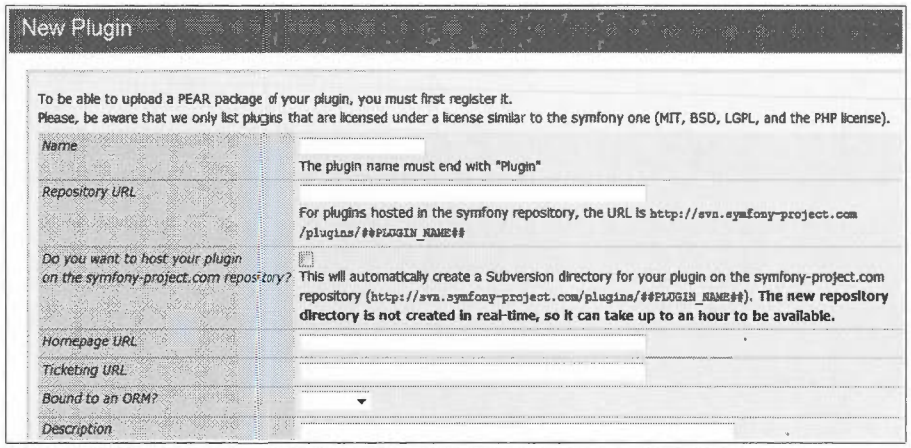


Figure 6.12 : Page de création d'un nouveau plugin (symfony)
<http://www.symfony-project.org/plugins/new> (consulté le 9 novembre 2011).

Cette figure est intéressante à analyser pour plusieurs raisons. D'abord, notons que les plugins doivent être publiés sous une licence similaire à celle du projet symfony, c'est-à-dire MIT, BSD, LGPL (nous reviendrons sur la question des licences à la section suivante). Ensuite, le nom du plugin doit se terminer par *Plugin*, en utilisant des noms tels que *sfDoctrineGuard**Plugin***, *sfFormExtra**Plugin***, *sfI18N**Plugin*** ou encore *idlErrorManagement**Plugin***. Ces noms font l'objet d'une contrainte que nous pourrions qualifier de « forte », en ce sens que si le nom du plugin ne se termine pas par *Plugin*, celui-ci ne fonctionnera pas[172]. Encore une fois, on voit bien ici que la performativité d'un morceau de code source n'est pas intrinsèque, mais dépend plutôt, pour reprendre cette expression de

171 <http://www.symfony-project.org/plugins/new> (consulté le 9 novembre 2011).

Fraenkel (2006), de son insertion dans une chaîne d'écriture plus large. Finalement, la description du plugin doit présenter différents éléments pour en faciliter la gestion, tels que l'URL (le lien Internet) du dépôt de versions (*repository*) où ce plugin sera affiché, de même que celui du « *ticketing* », c'est-à-dire du suivi de tickets.

6.3.3 Les plugins et la « Zone » de SPIP

> La devise par défaut lorsqu'il n'y a pas de fichiers expliquant une autre règle, c'est : on commit d'abord, on s'engueule après ! (liste spip-zone, 21 août 2009).

Dans SPIP, les modifications au code source sont plutôt validées a posteriori et la gestion des droits de commit s'appuie davantage sur des règles conventionnelles plutôt que sur des autorisations implantées dans le dispositif technique. Il existe deux dépôts distincts du code source dans SPIP. Le premier est réservé pour le « *core* » de SPIP. L'autre dépôt, nommé « La Zone », regroupe le code source des plugins et les squelettes de SPIP, entre autres choses. Contrairement à symfony où les droits d'accès sur les plugins sont délimités de façon « forte » par le dispositif technique, il n'y a dans SPIP qu'une seule gestion de l'accès pour chacun des deux dépôts, ce qui crée une situation « binaire » où il est seulement possible, ou non, de commiter :

> Donc, du coup on a une situation binaire où il y a des gens qui sont dedans et des gens qui ne sont pas dedans. Et ceux qui sont dedans peuvent commiter sur l'ensemble des fichiers. Et ceux qui ne sont pas dedans ne peuvent commiter sur rien (spip01).

Cette configuration s'explique probablement par la difficulté technique à configurer le système de gestion des versions de cette manière, mais également par un désir d'éviter une situation perçue comme potentiellement difficile à gérer « Si on dit, toi tu as le droit de commiter sur tel fichier, mais pas sur tel autre, etc. Ce serait débile en termes de gestion bureaucratique » (spip01).

Dans le cas de SPIP, les droits d'accès sur chacun des plugins s'appuient plutôt sur des règles conventionnelles. La première condition pour obtenir le droit de commiter à la *Zone* et, par le

---

172 Voir par exemple la conversation : « sfDoctrinePlugin2 won't work. The plugin name must end with Plugin » <http://groups.google.com/group/symfony-devs/msg/5403a67441f50172> (consulté le 2 mars 2012).

fait même, au développement de plugins est d'en obtenir les codes d'accès. C'est en fait la seule contrainte « forte » – c'est-à-dire régulée par le dispositif technique – qui délimite le droit de commiter. La page décrivant cette procédure présente les conditions pour obtenir ces codes d'accès : être intéressé par un projet spécifique à développer sur la Zone; s'inscrire à la liste de discussion spip-zone@rezo.net qui concerne les projets hébergés par la *Zone* et finalement, demander explicitement un accès en écriture à la *Zone*. Avant tout, la procédure indique comme première condition la nécessité de souscrire à la *Charte de la Zone*[173] (que nous incluons à l'appendice D). Cette Charte décrit notamment certaines valeurs du groupe auxquelles un contributeur doit adhérer, valeurs sur lesquelles nous reviendrons dans la prochaine partie. Au niveau de la coordination entre les contributeurs, la *Charte* précise qu'il est « obligatoire de respecter les termes des **règles d'intervention** définies dans les projets sur lesquels on intervient » (le gras est dans le texte original). Une autre page web de SPIP, celle des *Questions fréquentes*, précise ce que signifient ces règles d'intervention. Ainsi, dans la section intitulée « Droits d'intervention (et de commit) », on retrouve l'extrait suivant :

> Chaque branche (ou sous-branche, voire même fichier) de l'arborescence est **maintenue par une ou plusieurs personnes**, qui définissent les règles d'intervention **pour cette branche**. Ces règles peuvent éventuellement êtres *[sic]* fluctuantes dans le temps, au gré de la composition des équipes de dév, ou en fonction d'impératifs comme la stabilisation d'un projet, par exemple, ou pour toute autre raison.
>
> Il est *essentiel*, avant d'intervenir sur un fichier, de prendre connaissance des règles d'intervention liées à ce fichier, en remontant dans l'arborescence, à partir du fichier en question, jusqu'à trouver un fichier nommé **_REGLES_DE_COMMIT.txt' ; il est** obligatoire **de se conformer strictement à ces règles**[174]**.** (le gras est dans le texte original)

Ainsi, plusieurs répertoires de la *Zone*, mais pas tous, incluent ce fichier nommé _REGLES_DE_COMMIT.txt. Il s'agit en général d'un fichier par plugin. Comme indiqué sur la page des *Questions fréquentes*, les règles spécifiées dans ce fichier peuvent aller « d'une extrême liberté à un extrême besoin de contrôle[175]». La règle peut spécifier par exemple que chacun peut modifier ou ajouter des fichiers comme il le souhaite, ou à l'inverse, que les

---

173 <http://zone.spip.org/trac/spip-zone/wiki/CharteDeFonctionnement> (consulté le 8 novembre 2010).

174 <http://zone.spip.org/trac/spip-zone/wiki/FaQ> (consulté le 8 octobre 2011).

175 <http://zone.spip.org/trac/spip-zone/wiki/FaQ> (consulté le 8 octobre 2011).

modifications directes sont interdites, et qu'il est nécessaire d'envoyer une rustine à l'auteur du projet, ou du fichier. Comme le mentionne la citation placée au début de cette sous-section, s'il n'y a pas de fichiers de règles, la devise par défaut semble être « on commit d'abord, on s'engueule après ! ». Ceci occasionne des situations où, comme nous l'avons décrit dans notre étude de cas du commit 37817 de SPIP (section 6.2.2), un commit qui est jugé a posteriori comme étant inadéquat, doit ensuite être « dé-commité ».

Par ailleurs, notons l'inscription dans la *Charte* de l'obligation d'adhérer aux valeurs du projet, afin d'obtenir les droits de commit sur la *Zone*. Ensuite, la Charte insiste sur le respect des buts et des valeurs du projet SPIP, en mentionnant trois valeurs importantes, soit la promotion et la défense de la liberté d'expression sur Internet, la défiance vis-à-vis de l'argent et le respect de l'identité de chacun. Le premier item, dans ces valeurs, concerne toutefois la question du droit d'auteur et stipule que toutes les contributions sur la *Zone* doivent être libres « au sens de GNU », en utilisant par exemple des licences telles que la GPL, la LPGL ou la licence Art Libre.

Mentionnons pour terminer que cette obligation d'utiliser des licences libres « au sens de GNU » contraste de façon frappante avec l'approche de symfony, qui préconise au contraire l'utilisation de licences similaires à celles du cœur de symfony, soit les licences MIT, PHP, BSD ou LGPL (figure 6.12). En effet, c'est en analysant respectivement la *Charte* de SPIP et les plugins de symfony que nous avons découvert, avec surprise, que le code source de chacun des projets est souvent ***légalement incompatible***. Ainsi, alors que la GPL est la licence privilégiée dans le cadre de SPIP, celle-ci est carrément proscrite dans le cas de symfony. En analysant certaines discussions de symfony, on s'aperçoit que le choix de proscrire la licence GPL est justifié par le désir de favoriser le développement commercial[176]. Cette démarche va évidemment dans le sens contraire de la valeur pour le moins antagoniste de « défiance vis-à-vis de l'argent » stipulées dans la *Charte* de SPIP. On retrouve donc bien là une certaine articulation des valeurs de chacun des projets aux autorisations entourant les droits de commiter.

176 <http://www.mail-archive.com/symfony-users@googlegroups.com/msg02886.html> (consulté le 2 mars 2012).

6.4 Le commit comme contribution. Écologies de la visibilité et de l'autorité autour de l'acte du commit

Comme nous l'avons mentionné au chapitre 2, la *forme contribution* occupe une place de plus en plus grande dans la société contemporaine, au point où certains auteurs parlent de l'émergence d'une véritable « économie de la contribution » (Stiegler, Giffard, et Faure, 2009) ou considèrent que cette transaction serait « le fait social total d'un capitalisme cognitif et connecté, fortement irrigué par les médias sociaux » (Licoppe, Proulx, et Cudicio, 2010). Dans cette perspective, le commit occupe une place particulière. Il peut être appréhendé comme une contribution de plusieurs manières.

Le commit peut d'abord être appréhendé, de façon générale (et évidente), comme une contribution au dispositif technique. Cette perspective rejoint l'approche développée par Latzko-Toth (2010), pour qui la contribution sur Internet ne se limite pas à la production de contenu et de données, mais prend également la forme de contribution à la reconfiguration des dispositifs techniques. Latzko-Toth soutient que la contribution constitue une redistribution de la capacité d'agir, dans ce sens qu'elle reconfigure la capacité d'action des dispositifs techniques et celle des acteurs humains. De manière plus précise, le commit pourrait également correspondre à la notion d' « acte configurant » mise de l'avant par Latzko-Toth dans son étude de l'évolution du dispositif d'IRC, pour décrire une rustine (le « "*patch*" anti-Eris »), dont la fonction était d'ajuster les paramètres spécifiés dans le code et dans le fichier de configuration (Latzko-Toth, 2010, p. 319; voir également au chapitre 2).

La commit semble ensuite correspondre à la définition plus formelle que donnent Licoppe, Proulx, et Cudicio (2010) à la contribution, soit une transaction qui ne relèverait ni du don, ni de la transaction marchande, mais plutôt de « cette sorte de transaction fondée sur l'échange de messages isolables et discrets (une information, un nom, un numéro de version logicielle), orientée vers le service, l'entraide, la coopération interpersonnelle » (Licoppe, Proulx, et Cudicio, 2010). Cette perspective ressort particulièrement dans notre étude de cas du commit 20870 de symfony (section 6.2.1). Le commit y apparaît comme un acte couronnant une contribution qui implique un certain travail, formé d'autres transactions, telles que l'ouverture d'un ticket, la soumission d'une rustine ou la publication d'un commentaire, qui sont autant de transactions prenant elles aussi une « forme contribution ».

Le commit peut finalement être appréhendé comme une contribution parce qu'il est considéré ainsi par les acteurs. Pour illustrer l'importance du commit pour les acteurs, il est par exemple significatif de noter qu'un des acteurs de SPIP a choisi comme signature de ses courriels le pseudonyme *Committo, Ergo:sum,* faisant allusion à l'expression de Descartes, *Cogito Ergo:sum,* pour signifier « Je commit donc je suis ».

> C'est un jeu de mot avec Descartes, un jour, ça m'est apparu que *être*, dans la société moderne, c'est participer aux logiciels libres. À la question : qu'est-ce tu as fait de ta vie, est-ce qu'au moins tu as versé ton ego, en contrepartie à ton utilisation d'Internet, en apportant sur Internet un outil utilisable par les autres. Je commit donc je suis ! Il y a un coté très symptomatique de l'époque à laquelle on vit[177] !

Faisant référence plus spécifiquement à la notion de contribution, le site ohloh.net est également intéressant à analyser pour comprendre l'importance du commit parmi les acteurs. Les acteurs de chacun des deux projets étudiés nous ont en effet dirigés vers ce site. Dans le cadre d'une conférence publique, un des acteurs mentionnait même que de plus en plus d'employeurs utilisent ce site pour recruter des développeurs[178]. Ohloh.net est un service web dont le slogan est « Connect to people through the software you create & use » (traduction libre : Mettez-vous en contact avec les gens à travers le logiciel que vous créez et utilisez). Il s'agit d'un site web qui répertorie un grand nombre de projets de logiciels libres et qui propose différentes *metrics* – des indicateurs – pour analyser ces projets et dresser un profil de leurs contributeurs. Les indicateurs présentés sur ohloh.net sont établis à partir de l'analyse des dépôts de code source (tels que ceux basés sur Subversion), qui sont librement accessibles dans le cas des projets de logiciels libres. Parmi ces indicateurs, le nombre de commits occupe une place importante et permet de classer les « contributeurs » d'un projet donné (voir figure 6.13 plus bas)[179].

---

177 Nous sommes conscients que la référence au pseudonyme permet d'identifier l'auteur de ces propos. Nous préférons ici ne pas mentionner le numéro de l'entrevue pour conserver l'anonymat de notre interlocuteur dans ses autres propos.

178 Stefan Koopmanschap, « The symfony community - how you can (get) help ». Conférence *Symfony Live 2010*. Paris, 16 février 2010.

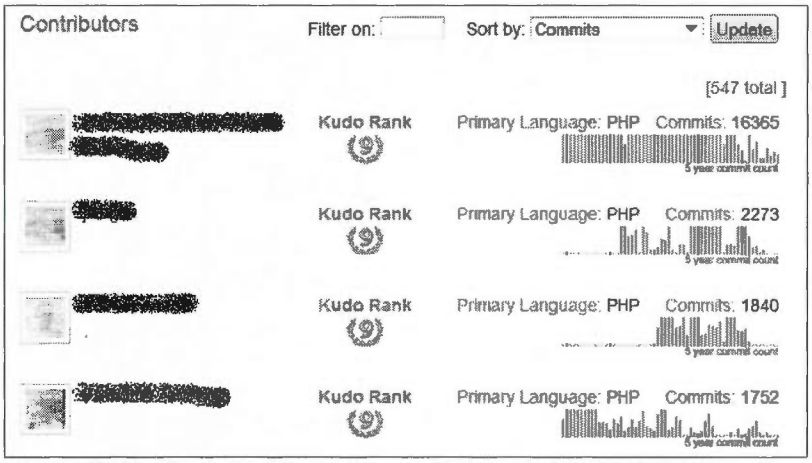


Figure 6.13 : Contributeurs et commits sur le site ohloh.net.

Cette importance accordée au commit comme forme privilégiée de contribution au code source implique toutefois une certaine « écologie de la visibilité[180] » (Star et Strauss, 1999) au sein des projets de logiciels libres. Ce sont ces enjeux de la visibilité que nous abordons dans la prochaine sous-section. La seconde sous-section aborde les questions d'autorité autour de l'acte du commit. Finalement, dans la dernière sous-section, nous présentons la technologie

179 Une autre problématique de ohloh.net est celle-ci plus collective, et concerne la division en branches. Comme nous l'avons mentionné plus tôt, SPIP n'organise pas la division entre plugins et *core* de la même manière que symfony. Dans le cas de SPIP, en effet, les plugins se situent dans *La Zone,* un répertoire SVN complètement distinct du répertoire du *core*. Dans le cas de symfony, au contraire, le *core* et les plugins se retrouvent dans le même répertoire, mais sont configurés selon des droits d'écriture distincts. Dans ohloh.net, SPIP et spip-zone sont donc répertoriés comme deux projets distincts, alors que le *core* et les plugins sont perçus comme un seul projet au sein de symfony. Or, comme nous l'avons mentionné plus tôt, la tendance actuelle au sein de SPIP penche plutôt vers une décentralisation du code, de façon à réduire les modifications au *core*. Ceci a pour effet de donner une apparence de réduction des contributions au sein du *core* de SPIP, alors même que la contribution augmente dans les plugins. Dans symfony, au contraire, on voit une contribution augmentant d'année en année. Ainsi, ohloh.net recense 502 contributeurs de symfony, alors que le *core team* compte moins d'une dizaine de personnes. Dans le cas de SPIP, au contraire, on retrouve 25 contributeurs pour le projet SPIP, et 272 contributeurs pour le projet spip-zone, soit le répertoire svn qui inclut notamment les plugins, les squelettes et la documentation (ces chiffres sont basés sur la consultation d'ohloh.net en date du 6 octobre 2010).

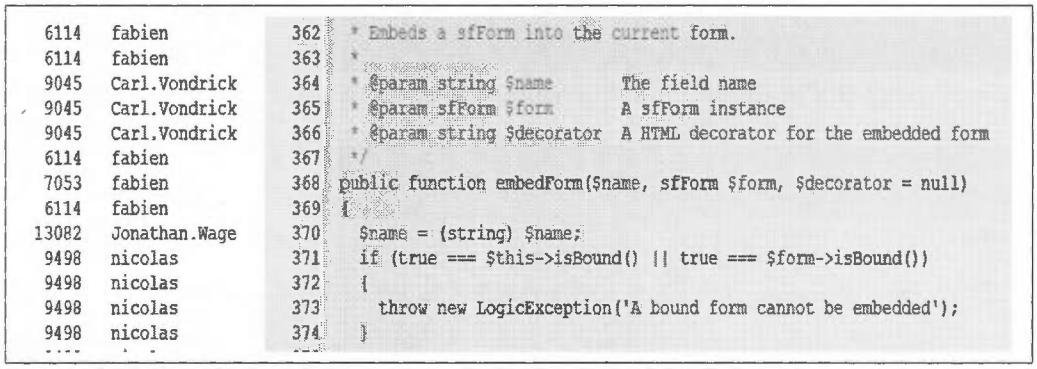
180 Dans cet article, les auteurs ne définissent pas explicitement la notion d'écologie. Ailleurs, Star (1995a) mobilise également la métaphore de l'écologie pour proposer le concept « d'écologie de la connaissance ». Critiquant la notion de *réseau* dans la théorie de l'acteur-réseau, Star explique dans ce texte préférer la métaphore de l'écologie pour insister sur le caractère « organique » de la connaissance (au contraire de celle de réseau qui suppose des discontinuités et des affiliations diverses).

GIT, désormais utilisée comme remplacement de SVN, et qui pourrait reconfigurer cette écologie de l'autorité et de la visibilité autour de l'acte du commit.

6.4.1 Écologies de la visibilité autour de l'acte du commit

Dans un article où ils analysent les relations entre travail visible et invisible, en particulier dans le contexte des dispositifs de travail coopératif assisté par ordinateur (*Computer Supported Cooperative Work* - CSCW), Star et Strauss (1999) notent qu'avec l'émergence des réseaux électroniques apparaissent de nouvelles manières de retracer et de valoriser le travail. Celles-ci sont souvent inscrites dans le langage « neutre » des métriques et sont particulièrement politiques, en ce sens qu'elles ont tendance à définir ce qui compte comme un travail à part entière. Comme Star et Strauss l'indiquent, le travail n'est jamais complètement visible ou invisible, mais est au contraire toujours « vu » à travers des indicateurs (Star et Strauss, 1999, p. 9). Le site ohloh.net, dont nous avons présenté une capture d'écran à la figure 6.13, montre bien qu'il en est de même pour la contribution : la contribution est toujours vue à travers certains indicateurs.

Un autre dispositif de visibilité qu'il est intéressant de mentionner concerne la commande *Praise* (Féliciter) du logiciel Subversion (le même ~~logiciel qui implante la commande~~ commit), qui nous a été décrite en entrevue (spip01). Cette commande permet d'identifier les auteurs de chacune des lignes du code source. La figure suivante (6.14) montre le résultat de cette commande pour une révision particulière d'un fichier de symfony. De la gauche vers la droite on retrouve : le numéro de la révision correspondant à cette modification au code source; le nom de « l'auteur » du commit; le numéro de la ligne, puis le contenu du code source à la ligne correspondante.



```
6114    fabien          362   * Embeds a sfForm into the current form.
6114    fabien          363   *
9045    Carl.Vondrick   364   * @param string $name       The field name
9045    Carl.Vondrick   365   * @param sfForm $form       A sfForm instance
9045    Carl.Vondrick   366   * @param string $decorator  A HTML decorator for the embedded form
6114    fabien          367   */
7053    fabien          368  public function embedForm($name, sfForm $form, $decorator = null)
6114    fabien          369  {
13082   Jonathan.Wage   370    $name = (string) $name;
9498    nicolas         371    if (true === $this->isBound() || true === $form->isBound())
9498    nicolas         372    {
9498    nicolas         373      throw new LogicException('A bound form cannot be embedded');
9498    nicolas         374    }
```

Figure 6.14 : Le résultat de la commande Praise du logiciel Subversion
(fichier http://svn.symfony-project.com/branches/1.4/lib/form/sfForm.class.php révision 2978).

Comme son nom l'indique, la commande *Praise* a pour objectif de féliciter les auteurs de chacune des lignes. Il est cependant possible d'obtenir le même résultat que la figure 6.14 en invoquant les commandes *Blame* (Blâmer) ou, de façon plus neutre, *Annotate* (Annoter). La symbolique de cet ensemble de commandes « synonymes » (dans le sens qu'elles donnent le même résultat) n'est pas anodine et dénote bien une certaine reconnaissance, de même qu'une certaine imputabilité, liée à l'acte du commit. Derrière cette reconnaissance, voire cette imputabilité du commit, une part importante du travail reste cependant invisible.

Dans l'étude de cas du commit de symfony que nous avons présentée à la section 6.2.1, nous avons par exemple décrit comment la rustine constitue une forme de contribution au code source. Toutefois, il est important de noter que, du moins dans le logiciel SVN (nous verrons plus tard que le logiciel GIT pourrait permettre de résoudre ce problème), la personne qui contribue sous la forme de rustines n'est généralement pas celle qui va réaliser le commit du code. En fait, nous pourrions même dire que si une personne contribue sous forme de rustines, c'est précisément parce qu'elle n'a pas suffisamment de droits pour contribuer sous forme de commits. Le résultat de cette situation est que les auteurs des contributions sous forme de rustines restent invisibles dans les contributions générales au projet. La reconnaissance de la contribution va donc plutôt à celui ou celle (mais, disons-le franchement, il s'agit généralement de « celui ») qui accomplit l'acte de commit, donc qui signe la contribution au code source.

Dans cette perspective, il est intéressant de constater le caractère central, voire même unique du commit dans ces indicateurs, au détriment d'autres indicateurs qui pourraient être

intéressants pour juger la contribution des gens. Un autre indicateur qui aurait pu être retenu par ohloh.net est le nombre de courriels envoyés pour chacune des listes, ce qui aurait offert une classification alternative des contributions des gens. Dans une très sommaire analyse quantitative, nous avons par exemple comparé le nombre de commits et le nombre d'interventions envoyées sur la liste spip-zone (tableau 6.1). Il apparaît que parmi les dix personnes qui ont commité le plus entre le 1er juillet 2009 et le 1er juillet 2010, seule la moitié figure parmi les dix personnes ayant envoyé le plus de courriels sur la liste. Par exemple, la seconde personne ayant réalisé le plus de commits n'intervient pratiquement pas sur les listes de discussions.

Tableau 6.1 : Nombre de commits réalisés sur la Zone et nombre de courriels envoyés sur la liste spip-zone[181]

Nb. de commits sur la *Zone*			Nb. de courriels sur la liste *spip-zone*	
1	**Martin**	1 270	**Martin (1)**	427
2	Frédéric	666	**Jérôme (8)**	258
3	**Stéphane**	504	Antoine	241
4	Gérard	442	**Stéphane (4)**	232
5	**Christian**	399	**François (10)**	214
6	Jonathan	373	**Christian (5)**	148
7	Patrick	314	David	126
8	**Jérôme**	314	Serge	124
9	Mathieu	300	Sébastien	114
10	**François**	252	Jack	104

La réalisation d'entretiens et l'observation, outre que d'aider à mieux comprendre le projet dans son ensemble, permettent également de saisir d'autres formes de travail, invisibles ou moins directement visibles par la seule analyse des traces en ligne. Dans chacun des projets étudiés, on remarque en effet une multitude de rencontres présentielles, qui prennent en particulier la forme de séances d'entraide plutôt informelles dans SPIP, et de formation dans le cadre de symfony. D'ailleurs, l'une des personnes que nous avons rencontrées (une femme), qui n'a effectué aucun commit mais dont le rôle est pourtant jugé important dans SPIP, nous disait que la plupart des gens actifs au sein de SPIP sont continuellement présents sur IRC. Dans le cas de symfony, l'entreprise qui commandite le projet emploie elle-même plusieurs personnes, qui peuvent donc se retrouver au quotidien et participer de différentes manières au

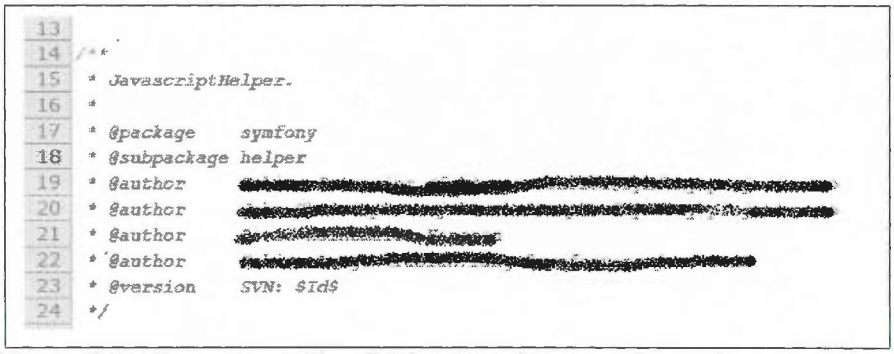
181 Note : les noms ont été changés pour conserver l'anonymat des acteurs.

projet. Dans un entretien, une participante nous explique ainsi qu'au moment de son embauche, elle avait suggéré au chef de projet de symfony d'utiliser une nouvelle version du langage PHP, ce qui, selon elle, a probablement contribué à ce choix. Un autre acteur explique ainsi sa contribution au projet symfony :

> Ma contribution, ma petite contribution dans symfony, ça a plus été de fédérer et monter une communauté de développeurs motivés plutôt que de participer par exemple au code et à la génération de plugins, où je me sens moins à l'aise (sf10).

D'une certaine manière, ces deux types de travail, « fédérer une communauté » et « participer au code », pourraient correspondre à la distinction faite par plusieurs auteurs et mise de l'avant par Star et Strauss, entre le travail d'articulation et le travail de coopération. Pour ces auteurs, le travail d'articulation – fédérer une communauté – consiste à gérer la nature distribuée du travail, tandis que le travail de coopération – participer au code – entrelace des tâches distribuées (Star et Strauss, 1999, p. 10). Le plus significatif pour ces auteurs est cependant que le travail d'articulation reste invisible aux modèles rationalisés du travail. L'important ici est donc de rappeler que si le commit constitue un acte central conférant une stabilité au code source, cet acte lui-même repose sur de multiples mains dont le travail présente différents degrés de visibilité. Si l'ouverture d'un ticket ou la proposition d'une rustine sont moins visibles que le commit, ils sont néanmoins des actes isolables qui peuvent être analysés en tant que tels. La question qui se pose est cependant : comment analyser le travail beaucoup moins visible d'articulation en regard de la stabilité du code ?

Par ailleurs, mentionnons pour terminer que les relations entre les notions d'auteurs et de contributeurs semblent être également articulées avec certaines dynamiques de visibilité. Ainsi, à la lecture des différents fichiers du code source, on peut constater que les premières lignes des fichiers contiennent une mention des auteurs liés à telle ou telle partie du code source. La figure 6.15 montre le nom des auteurs de la partie du code source concernée par le commit 20870. À la lecture de cette liste d'auteurs, on peut constater que la plupart des personnes qui ont participé au travail ayant conduit au commit 20870 ne figurent pas parmi les auteurs du commit en question. D'une certaine manière, nous pouvons dire que leur travail reste invisible.



```
13
14 /**
15  * JavascriptHelper.
16  *
17  * @package    symfony
18  * @subpackage helper
19  * @author     [redacted]
20  * @author     [redacted]
21  * @author     [redacted]
22  * @author     [redacted]
23  * @version    SVN: $Id$
24  */
```

Figure 6.15 : Les auteurs d'un fichier du code source de symfony
(fichier concerné par le commit 20870).

Dans le cas de SPIP, les auteurs sont également mentionnés dans le haut de chacun des fichiers du code source. Il est cependant remarquable de noter que certains des auteurs mentionnés sont des acteurs historiques plutôt que des acteurs encore actifs au sein du projet. On voit ici qu'auteur du commit et auteur du code source sont des concepts assez distincts.

6.4.2 Autorité et droits de commit

Le commit a déjà été abordé dans quelques études portant sur les logiciels libres, souvent pour insister sur le statut que confèrent les droits de commiter dans les projets donnés. Roberts *et al.* (2006) notent à propos des projets liés au logiciel *Apache* qu'il existe différentes manières de reconnaître le travail dans cette communauté, s'exprimant par différents statuts, dont celui de « *commiters* »[182]. Auray (2007)(2007), dans une étude sur la communauté FreeBSD, désigne les *commiters* comme des « membres agréés de la communauté » en indiquant qu'un vote de la majorité des *commiters* peut entraîner l'éviction d'un membre de l'équipe de coeur (le « *core team* »). Scacchi (2007) note également que l'acte du commit est étroitement associé à des droits et des privilèges concernant la possibilité de modifier le code source du projet. Finalement, dans leur travail sur le projet SPIP, Demazière, Horn et Zune (2006) avaient déjà noté l'accès différencié aux droits de commit qui entraîne certains rapports hiérarchiques entre les acteurs du projet : « une fois accordés, ces droits [de commit] ont la force d'un mandat, celui de poursuivre le développement en

182 Les différents statuts mentionnés par Roberts *et al.* (2006) sont, en ordre d'importance : *Développeur, commiter, membre du comité de gestion de projet* et *membre de la fondation Apache*.

toute autonomie, et ne s'obtiennent qu'après avoir fait la preuve d'une grande qualité et fiabilité techniques » (Demazière, Horn, et Zune, 2006, p. 10)

Si les droits en écriture sur le dépôt du code source concernent avant tout la possibilité de publier le code source « autorisé » (à travers notamment l'acte du commit), ces droits constituent également – et peut-être surtout – un marqueur d'autorité au sein du projet. Ainsi, dans SPIP, ceux et celles qui ont effectivement des droits de commiter sur le « *core* » sont de fait membres du « *core team* ». Cependant, il est important de mentionner que l'octroi des « droits de commiter » n'est pas nécessairement corollaire de l'activité de commiter. L'un des aspects intéressants que nous avons constatés, en particulier dans le projet SPIP, est que l'autorisation de commiter est parfois accordée à des personnes qui n'ont en fait que très peu, voire pas du tout, réalisé de commits. Ces personnes sont plutôt cooptées dans l'équipe de coeur parce que le rôle d'animateur, de modérateur ou d'administrateur qu'ils ou elles assument est jugé essentiel à la cohésion de la communauté ou au développement du logiciel[183]. Si les droits différenciés de commit ont pour effet pragmatique de réguler la coordination dans l'écriture du code source, ces droits agissent également, à titre symbolique, comme des marqueurs d'inclusion, voire d'autorité, au sein des projets.

Un dernier aspect que nous n'avons pas beaucoup exploré dans nos entrevues, mais qu'il nous semble important d'aborder ici, au moins brièvement, concerne une autorité entourant les dispositifs matériels de collaboration, et entre autres, le contrôle des serveurs. Cette autorité s'exprime explicitement dans les deux projets, notamment sur la question des plugins où l'on demande aux acteurs de ne pas héberger les plugins sur les serveurs des projets, s'ils ne souscrivent pas à la *Charte*. Dès lors, une question qui pourrait être importante est : qui contrôle(nt) les serveurs ?

Encore une fois, on peut distinguer symfony et SPIP par la manière dont cette autorité est centralisée dans le premier cas, et distribuée dans le second. Ainsi, dans symfony, le contrôle des serveurs revient en dernière instance à *Sensio Labs*, l'entreprise commanditaire de symfony. Dans le cas de SPIP, le code source du cœur semble être hébergé sur le serveur de

183 Dans le cas de symfony, nos entrevues ne nous permettent cependant pas de soutenir cette même affirmation.

l'un des fondateurs, tandis que la *Zone* est hébergée et administrée par des membres de la communauté SPIP qui ne font par ailleurs pas partie de l'équipe de coeur.

Toutefois, il est important de relativiser le pouvoir que peut conférer ce contrôle des serveurs. Dans le cas de SPIP en particulier, ce pouvoir est en effet balancé par d'autres mécanismes, en particulier la mise en place de miroirs :

> Mais en fait, ce n'est pas tellement ça qui confère un pouvoir parce que si demain il n'est pas d'accord, il y a des miroirs, moi j'ai un miroir, etc. Et donc, on peut *switcher* et continuer de développer sur un SVN à côté. Après il y a un *fork*. En gros, ça revient à un *fork* (spip11).

Le contrôle « technique » des serveurs participe donc d'un pouvoir mais ce pouvoir lui-même ne peut être restreint à ce seul contrôle technique. Ainsi, si la personne qui contrôle le serveur décide de fermer le serveur, il y aurait certainement une commotion dans le projet, mais celle-ci ne serait peut-être pas fatale. Le projet repose en effet en dernière instance sur la communauté de ses développeurs, traducteurs et utilisateurs qui décideront en dernier lieu à quelle autorité ils adhéreront :

> Mais le problème, il n'est pas là. Le problème, c'est de savoir que si il y a un *fork*, tout d'un coup ça devient un projet où il y a deux branches. La question, c'est où vont les utilisateurs et plus encore, je pense qu'une très grosse partie de l'enjeu est là, c'est où vont les traducteurs aussi (spip11).

Cette dernière citation montre bien les limites de notre analyse, car l'autorité du code semble reposer finalement en dernière instance sur les humains qui forment la communauté de ses développeurs, utilisateurs et traducteurs.

6.4.3 Ouverture : GIT, écriture distribuée et réseau social basé sur le code

Durant la période de notre enquête de terrain, les deux projets se sont progressivement tournés vers un nouveau système de gestion des versions, le logiciel *GIT*. Ainsi, la version 2.0 de Symfony (avec un « S » majuscule cette fois-ci !), annoncée en février 2010, est désormais gérée à la fois dans Subversion et dans GIT. De la même manière, SPIP s'est tourné vers GIT à la toute fin de notre enquête de terrain, pour gérer les versions du code source du « *core* ». Comme symfony, SPIP continue également d'utiliser Subversion. Ce « tournant GIT » s'étant réalisé dans une étape tardive de notre enquête, nous n'analysons pas ici les usages de GIT comme nous l'avons fait plus tôt pour SVN. Toutefois, il nous semble important de présenter

cette technologie et son importance pour les acteurs. En effet, autant dans les entrevues que nous avons réalisées que dans les conférences et rencontres auxquelles nous avons participé (comme observateur), GIT est revenu à plusieurs reprises, sans que nous abordions nous-mêmes le sujet. En fait, nous pourrions dire que la technologie GIT est quelque chose qui a émergé dans notre étude, en ce sens que c'est dans la rencontre avec les acteurs que nous avons pris connaissance de cette technologie (au contraire de Subversion, que nous connaissions avant d'entreprendre l'enquête). Nous présentons donc ici la technologie GIT, mais en nous appuyant surtout sur les propos des acteurs que sur l'analyse empirique de son usage (ce que nous avions fait pour SVN).

Pour plusieurs des acteurs que nous avons rencontrés, GIT présente des avantages importants par rapport à Subversion. Pour l'un des acteurs, « GIT a cet avantage de pouvoir favoriser le travail communautaire, qu'on ne peut pas véritablement faire avec Subversion » (sf07). C'est en effet pour sa simplification des tâches liées à l'application de rustines, que nous avons présentées à la section 6.2.1, que GIT serait à privilégier :

> Avec GIT, ils vont avoir directement la modification, ils vont voir si elle correspond, et puis, ils vont dire si oui ils l'appliquent, non ils ne l'appliquent pas, et GIT va gérer tout seul le *merge*[184], sans casser. Alors que dans Subversion, le processus est beaucoup plus long parce qu'il faut aller voir les tickets qui contiennent des *patches*, il faut appliquer le *patche*, vérifier que c'est bon, etc., re-commiter la modification (sf07).

Un autre acteur de symfony met de l'avant que « c'est plus facile d'échanger du code maintenant qu'avant » et que « GIT à mon avis a joué un rôle là-dedans » (sf06)[185]. Pour celui-ci, la supériorité de GIT par rapport à SVN (Subversion) est de raccourcir les tâches nécessaires à l'application d'une rustine :

---

184 *Merge* : le fait de fusionner deux versions du code source. Dans ce cas, fusionner une proposition de modification au code source (une rustine) avec le code source central.

185 D'un point de vue méthodologique, mentionnons ici que notre interlocuteur a tenu ces propos à la toute fin de l'entrevue, au moment où nous lui demandions s'il souhaitait aborder un point que nous n'avions pas discuté. Ceci montre à notre l'avis l'importance de GIT pour cet acteur.

> C'est-à-dire qu'avant, quand on voulait participer dans un logiciel libre, il fallait aller télécharger le code source, le modifier sur sa machine et puis envoyer un mail aux responsables du logiciel pour lui dire, voilà, j'ai écrit un *patche* qui fait ceci, cela, est-ce que vous pouvez l'intégrer. Donc la personne va le tester, et après c'est elle qui va le remonter dans le SVN. Donc, aujourd'hui avec GIT, et GitHub, qui est un service d'hébergement de projets, donc toutes ces tâches elles ont été énormément raccourcies en fait (sf06).

Dans SPIP, la technologie GIT nous a également été mentionnée dans quelques entrevues. Cependant, pour au moins un acteur, les raisons qui justifient le changement vers GIT semblent moins utilitaristes – simplifier les tâches – et plus politiques. Selon cet acteur, GIT permettrait d'éviter la configuration actuelle des droits de commit, où certaines personnes possèdent des droits et d'autres non. GIT favoriserait au contraire une plus grande ouverture du code source, en ne nécessitant pas de démarquer ceux qui sont « dedans » et ceux qui sont « dehors » :

> Je pense que la contrainte technique de SVN, elle n'est pas bonne pour notre projet. Et là, j'ai beaucoup regardé un système de gestion de versions qui s'appelle GIT, qui a été développé par Linus Torvalds, et GIT, c'est totalement différent comme fonctionnement. Avec GIT, il n'y a pas dedans, donc il n'y a pas de dehors. On ne peut pas dire que tout le monde est dedans, ou que tout le monde est dehors, c'est juste différent. Et je pense que ça serait un meilleur outil de gestion de SPIP que SVN (spip01).

Qu'est-ce que GIT ? Comme l'extrait d'entrevue précédent le mentionne, GIT a été développé par Linus Torvalds, le fondateur de Linux. La principale innovation de GIT semble être de mettre de l'avant une approche beaucoup plus distribuée de la gestion des versions que Subversion. Mais qu'est-ce que GIT et pourquoi, selon les acteurs, favorise-t-il le travail communautaire ? Le site web de GIT (par ailleurs uniquement en anglais) décrit le logiciel comme un « free & open source, distributed version control system »[186]. GIT constitue donc un système de contrôle des versions, tout comme Subversion et d'autres auparavant. Toutefois, GIT met de l'avant un caractère « distribué », au contraire des logiciels plus anciens tels que SVN (ou encore CVS). L'une des principales innovations de GIT par rapport à SVN – du moins selon les acteurs – semble être d'avoir éliminé la notion de dépôt central, au profit d'une approche, où chaque utilisateur de GIT possède son propre dépôt de code source :

---

186 <https://github.com/> (consulté le 1er novembre 2011).

> Dans GIT, tout est décentralisé. C'est-à-dire que, dans Subversion, le dépôt est sur un serveur, il est centralisé. Dans GIT, l'approche est différente, c'est-à-dire que chaque copie de travail est un dépôt de suivi de versions. Donc on peut commiter dans son propre dépôt local, et puis à la fin lui dire, voilà je commite dans un dépôt qui est distant. GIT, grâce à cette approche qui est légèrement différente, facilite énormément le travail communautaire (sf07).

En d'autres termes, lorsqu'un contributeur souhaite apporter une modification au code, sans toutefois disposer des droits de commit, il fera d'abord un commit dans son dépôt local, puis « diffusera » le commit dans le dépôt autorisé, tout en étant reconnu comme auteur du commit. L'une des principales conséquences de cette innovation, du moins selon le point de vue de notre recherche, est de rendre caduque la notion de rustine (« *patche* »), sur laquelle nous avons beaucoup insisté plus tôt dans ce chapitre, en particulier dans l'étude de cas de symfony. C'est notamment ce qui « favorise le travail communautaire » (sf07) en diminuant le temps consacré à chacune des contributions. Ainsi, une opération qui pouvait prendre auparavant une journée complète ne requiert désormais que quelques transactions élémentaires. Un acteur décrit ainsi la simplicité des opérations dans GIT :

> C'est-à-dire, que moi, mon plugin Solar machin, est sur GITHub. Et si quelqu'un en fait veut le modifier, et bien en un clique, il peut récupérer l'ensemble du code source sur son propre compte. En une commande, il peut le récupérer sur sa machine. Il peut faire ses développements [...] et puis après, en un clique, il peut me dire voilà j'ai fait tel truc, regarde ce que j'ai fait. En gain de temps, on peut échanger des bouts de code comme ça alors que ça aurait pris une journée avant (sf06)[187].

Le site GitHub, mentionné dans cet extrait, est également important à souligner. GitHub est un service web qui, comme son nom l'indique, est étroitement associé la technologie GIT. Les acteurs du projet le décrivent comme « un service d'hébergement de projets » (sf07), voire « presque un réseau social qui est bâti sur le code » (sf06). Le page principal de GitHub[188] présente le service comme un site de « social coding » qui permet de gérer tous ses dépôts GIT. En plus des dépôts GIT, le service GitHub propose aux usagers des outils de collaboration tels que le suivi des « *issues* » (des « problèmes », comme les bogues ou les

---

187 Cette description est sans doute un peu idéalisée et ne tient certainement pas compte du temps nécessaire à l'appropriation du logiciel GIT et de la structure des fichiers visés par le commit. En effet, nous avons nous-mêmes tenté d'utiliser GIT. Après un certain temps, nous avons décidé d'abandonner cet essai n'ayant pu comprendre le fonctionnement de cette technologie.

188 <https://github.com/> (consulté le 1er novembre 2011).

demandes de nouvelles fonctionnalités), des wikis ou des pages de téléchargement (download pages), entre autres choses. Au moment d'écrire ces lignes, la page principale du site GitHub regroupe plus d'un million d'usagers et trois millions de dépôts GIT[189] (le dépôt du code source de symfony, et celui de SPIP étant chacun un dépôt de code source sur GitHub). Le site GitHub montre bien la manière dont le code source circule et évolue aujourd'hui dans une dynamique très similaire à celles des autres médias de type web 2.0.

Mentionnons finalement l'importance grandissante de la notion de contributeur dans le cas de GIT, et la relation de cette notion avec celle d'auteur. En effet, la page « About » du site GIT[190] (à ne pas confondre avec celle du site GitHub) décrit les différents auteurs primaires *et* les contributeurs de GIT (qui sont par ailleurs qualifiés de « héros »), spécifiant qu'ils sont regroupés par nombre de commits. Si cette évolution peut paraître plus égalitaire et donner plus de reconnaissance aux différentes formes de contributions, on peut également constater la centralité grandissante du commit dans la reconnaissance de cette contribution.

6.5 Conclusion partielle : le commit comme acte de reconfiguration

Dans le premier chapitre, nous avons abordé la manière dont certains auteurs, en particulier Lessig (2006), abordent le code informatique à la manière d'une loi qui prescrit ou contraint les comportements humains. Dans un article plus daté, Lessig (1998) propose une réflexion sur l'autorité sur Internet, en mettant de l'avant une catégorie qui nous semble pertinente ici : le *running code*. Dans cet essai, Lessig soutient que le mouvement du logiciel libre minimise l'autorité des règles formelles, par exemple celle des rois, des présidents et du vote, au profit d'une autre forme d'autorité, celle du consensus approximatif provenant du code exécutant, « this is not to say the Net rejects authority. Indeed [it] identifies the source of authority — "rough consensus and running code" — and between the two, I suggest, it is the second that is more important ») (Lessig, 1998, p. 114). Évidemment, cet article étant relativement daté, plusieurs études ont depuis mis de l'avant la complexité des dynamiques d'autorité et de reconnaissance sur Internet et il est probable que Lessig lui-même nuancerait son idéalisme d'alors. Cet article de Lessig permet toutefois de saisir le caractère infrastructurel, voire

189 <https://github.com/> (consulté le 1er novembre 2011).

190 <http://git-scm.com/about> (consulté le 1er décembre 2010).

matériel, du code informatique sous sa forme de « running code ». Le « running code » chez Lessig, a une force pragmatique : il constitue l'architecture de l'Internet, qui prescrit ou limite certains comportements. Une question se pose toutefois : s'agit-il simplement d'écrire du code (et de l'écrire bien) pour que celui-ci s'exécute ?

Les descriptions et analyses réalisées dans ce chapitre montrent bien que la situation est plus complexe. Nous avons en effet vu que le « code source autorisé », celui qui est disponible au téléchargement et qui constituerait ce que Lessig appelle le « running code », est étroitement articulé à différentes autorisations qui prennent par exemple la forme de droits de commit. En d'autres termes, il ne s'agit pas seulement d'écrire un morceau de code source pour que celui-ci « performe ». La performativité d'un morceau de code source dépend au contraire en bonne partie, pour reprendre les termes de Fraenkel, de son insertion dans « un système de chaînes d'écriture, de personnes habilitées et de signes de validation, éléments qui forment l'authenticité nécessaire à la performativité » (Fraenkel, 2006). Dans ces dynamiques, le commit a une pragmatique particulière, car c'est cet acte qui entérine l'insertion d'un morceau de code source, ou d'une proposition de modification au code source, dans la chaîne d'écriture plus large que constitue le code source.

À bien des égards, le commit pourrait donc être comparée à l'acte de la signature, amplement discuté par Fraenkel. Cette comparaison du commit avec l'acte de la signature a cependant ses limites. D'une part, comme nous l'avons noté dans les chapitres précédents, si le code source prend bien souvent la forme d'un écrit, ce n'est pas toujours le cas et celui-ci doit, par conséquent, être défini de façon plus large qu'un écrit (voir à ce propos la note de base page nu. 160, page 213). D'autre part, alors que la signature confère à l'acte juridique l'authenticité nécessaire à sa performativité, l'acte du commit a plutôt comme conséquence pragmatique – du moins dans le contexte de Subversion – d'insérer un morceau de code source dans un ensemble plus grand de code source dont le statut et la performativité diffèrent également en fonction de l'endroit où il se situe. En d'autres termes, la performativité du code source diffère de celle de l'acte juridique qu'analyse Fraenkel, car la performativité du code source, au contraire de l'acte juridique, ne dépend pas tant de son « authenticité », mais plutôt de l'endroit où celui-ci est situé. D'une certaine manière, dans le cas du code source, il vaudrait peut-être mieux parler d'une force performative plus ou moins grande, plutôt que du simple échec ou du succès de la performativité. Ce qui est important ici, toutefois, c'est l'articulation

de la performativité d'un morceau de code source en regard du « lieu » où celui-ci est inséré, et le rôle du commit dans cette insertion.

C'est probablement ce rôle important du commit en regard de la performativité du code source qui fait que cet acte est étroitement articulé à des dynamiques d'autorité et d'autorisation, ainsi qu'à une écologie de la visibilité. Toutefois, les autorisations qui participent à l'acte du commit ont également des conséquences politiques. D'une part, comme nous l'avons mentionné, ces autorisations sont articulées aux valeurs des acteurs et des projets respectifs. Ainsi, dans le cadre de SPIP, l'obtention des droits de commiter implique l'adhésion aux valeurs non commerciales spécifiées dans la *Charte*. Bien que symfony ne présente pas ses valeurs de façon explicite, le droit de commiter interdit de commiter du code source qui serait soumis à une licence compatible avec la GPL.

Par ailleurs, si le commit est perçu comme une forme particulièrement privilégiée de la contribution, celui-ci repose souvent sur un important travail de petites mains qui s'exprime par exemple dans la soumission de rustines, voire même d'un travail d'articulation, beaucoup moins quantifiable. Ainsi, dans le cas des projets étudiés, le commit est articulé avec différentes dynamiques de visibilité et opère certains découpages. Dès lors, pour revenir à notre analyse du code source et du commit, l'enjeu à la fois théorique et politique consisterait à saisir des actes qui, sans nécessairement être des actes d'écriture du code source proprement dit, seraient néanmoins des actes qui participeraient à la stabilité et à la vitalité du code source, et qui permettraient de saisir un éventail d'activités plus large.

CHAPITRE VII

CONCLUSION GÉNÉRALE

> More than conversation at the interface, we need the creative elaboration of the particular dynamic capacities that these new media afford and of the ways that through them humans and machines together can perform interesting new effects (Suchman, 2007, p. 23).

Dans ce chapitre de conclusion, nous exposons de façon synthétique les principaux éléments qui ressortent de la thèse. Nous tentons d'abord de répondre explicitement à chacune des questions de recherche que nous avions formulées au premier chapitre. Nous faisons ainsi ressortir le flou définitionnel entourant la définition de code source, les mécanismes qui participent à la stabilisation de cet artefact et de sa performativité, ainsi que la manière dont le code source est traversé de valeurs et de rapports d'autorité. Finalement, nous proposons une conceptualisation théorique du code source, en tant qu'interface des reconfigurations humain-machine, en insistant sur la manière dont ces interfaces sont conçues et « découpées », ce qui entraîne certaines conséquences, en particulier en favorisant la participation de certains acteurs plutôt que d'autres. Deux pistes futures de recherche ressortent également de notre étude : 1) l'étude de l'inscription de valeurs dans le design du code source et 2) l'analyse d'autres formes d'écriture « humain-machine » qui mélangent de plus en plus étroitement langage naturel et langage structuré, comme c'est le cas par exemple des wikis, qui intègrent des codes de formatage au langage naturel.

7.1 Retour sur nos questions de recherche

Il nous semble ici pertinent de revenir sur nos questions de recherche. Rappelons que notre question centrale était : *Qu'est-ce que le code source et comment cet artefact agit-il dans les reconfigurations d'Internet ?* Nous avions également posé quatre questions spécifiques nous aidant à orienter notre démarche de recherche :

1) Comment les acteurs définissent-ils le code source et en particulier, dans quelle mesure cet artefact peut-il être appréhendé comme un écrit ?

2) Comment la fabrication et la stabilisation du code source s'articule-t-elle à sa force « législative », ou performative ?

3) De quelles manières certaines valeurs sont-elles inscrites dans la fabrication et le design du code source ?

4) Comment le code source agit-il dans les reconfigurations d'Internet ?

Nous cherchons dans les sous-sections qui suivent à répondre plus explicitement à chacune de ces questions de recherche.

7.1.1 Le code source, un artefact aux frontières ambiguës

Comment les acteurs définissent-ils le code source et en particulier, dans quelle mesure cet artefact peut-il être appréhendé comme un écrit ?

Notre étude a permis de faire ressortir un certain flou entourant la définition de « code source », et ce autant dans la littérature que dans les propos des acteurs qui ont participé à notre enquête. De façon commune, dans le discours des acteurs, le code source est souvent appréhendé comme un « texte », quelque chose que l'on écrit. De fait, concrètement, nous avons vu que le code source prend la plupart du temps la forme d'un texte, qui peut être édité à partir d'un éditeur de texte (tel que *Notepad* par exemple). Une analyse plus poussée des définitions données à la notion de code source ainsi que des différentes formes que prend cet artefact dans les projets étudiés, montre cependant que les frontières définitionnelles de cette notion sont encore aujourd'hui ambiguës. Ainsi, si le code source peut être généralement appréhendé comme un artefact écrit, ce n'est cependant pas toujours le cas.

En effet, ce que les acteurs désignent par le terme de code source renvoie souvent à différents types d'objets, comme des images, ou encore des textes qui ne sont pas directement destinés à faire fonctionner l'ordinateur, mais qui permettent, par exemple, de produire de la documentation technique. Un acteur (spip01) mentionne même que le code source, dans le cas d'Internet, serait les *Request For Comments* (RFC), qui sont des spécifications formelles

de certaines composantes d'Internet, écrites en langue informatique, plutôt qu'un « code informatique, au sens de langage de programmation » (spip01).

Ce flou définitionnel entourant la notion de code source est sans doute l'un des aspects les plus étonnants de notre étude. La manière dont nous avons présenté cet aspect, en l'abordant dès le début de la problématique, pourrait laisser croire que nous avions déjà cette intuition dès le début de la thèse. Or, ce n'est pas le cas. Étant diplômé en informatique (au premier cycle) et ayant une certaine expérience de programmation, nous avions, en début de thèse, une idée assez claire de ce qui constituait le code source. Le caractère problématique du code source est apparu plus tard, sous l'insistance de nos directeurs de nous attarder davantage aux perspectives des acteurs pour définir le code source[191]. C'est ensuite en discutant avec un acteur du projet SPIP qui nous affirmait que le code source n'existait pas dans le cadre du projet SPIP, que nous avons décidé d'approfondir les définitions du code source, autant chez les acteurs que dans la littérature. Les résultats de cette démarche ont été présentés dans un premier temps en problématique, par une brève analyse de la définition de code source. Nous avions par exemple montré comment la licence publique générale GNU (GPL) mettait de l'avant une définition très générale du code source comme étant la « forme privilégiée du travail pour réaliser des modifications ». Nous avons ensuite consacré le chapitre 4 à l'analyse des différentes définitions du code source données par nos acteurs, ainsi qu'à l'analyse des multiples formes et statuts du code source dans les projets étudiés. Ce chapitre empirique faisait également état d'une certaine difficulté à définir, clairement et de façon circonscrite, la notion de code source.

La difficulté à définir la notion de code source semble confirmée par certains travaux dont nous avons pris connaissance dans une étape tardive de l'étude, et qu'il nous semble pertinent de présenter ici. Ainsi, Robles et Gonzalez-Barahona (2004), analysant le dépôt de code source de KDE, notent que le code source, entendu comme les composantes utilisées pour produire une version exécutable d'un logiciel, comprend davantage que la conception « classique » (c'est le terme qu'ils utilisent) du code source en tant que texte écrit dans un langage de programmation. Les auteurs notent, comme nous l'avons nous-mêmes constaté

191 Merci en particulier à Serge Proulx qui a insisté sur l'importance de consacrer un chapitre complet aux définitions que les acteurs donnent au code source.

dans SPIP et symfony, que plusieurs autres types de fichiers sont constitutifs du code source. Ces fichiers comprennent par exemple de la documentation technique, des spécifications de l'interface, des modules de traduction et des fichiers multimédias, notamment des images. Plus important encore, en ce qui concerne nos propos, Robles et Gonzalez-Barahona notent que ces fichiers qui ne sont pas composés de langages de programmation, *prennent une place de plus en plus importante dans le cas des applications finales* (*end-user applications*) :

> The concept of source code, understood as the source components used to obtain a binary, ready to execute version of a program, comprises currently more than source code written in a programming language. Specially when we move apart from systems-programming and enter the realm of end-user applications, we find source files with documentation, interface specifications, internationalization and localization modules, multimedia files, etc. All of them are source code in the sense that the developer works directly with them, and the application is built automatically using them as input (Robles et Gonzalez-Barahona, 2004, p. 1).

Ce constat devrait nous amener, dans le cadre des études en communication portant sur les technologies de l'information, à considérer les différents formes que peuvent prendre ces composantes utilisées pour produire des applications finales, et en particulier, à considérer leurs conséquences sur la capacité des usagers à participer à la production, ou la reconfiguration des applications finales. Cette perspective, qui est développée davantage à la section 7.1.4 (Le code source comme interface dans les reconfigurations d'Internet), constitue le cœur de notre thèse.

Cette conception étendue du code source a des conséquences autant en termes politiques qu'en termes de design. D'abord, en termes politiques, nous l'avons vu, la catégorie du « code » est problématique pour quelques-uns des participants. Nous avons ainsi mentionné la manière dont, dans SPIP, les contributions semblent valorisées différemment selon qu'il s'agit ou non d'une contribution sous forme de code source (ou du moins, perçue comme telle par les acteurs/actrices). Cette valorisation du code s'exprime également dans nos entrevues par la mention de la catégorie du « codeur », ce qui montre bien une certaine hiérarchisation de l'activité, en fonction de sa proximité avec le code (source). Plus on s'approche de ce qui est considéré comme du « vrai » code, en particulier le code source du « coeur » de chacun des projets, plus l'activité semble être valorisée. Cette valorisation de certains types d'activités liées au code source se retrouve également dans d'autres contextes. Dans notre recension des

écrits, nous avons par exemple mentionné la recherche de Ghosh *et al.* (2002), où avait été empiriquement circonscrite la catégorie du « développeur de logiciel » aux seuls individus qui figuraient comme « auteurs » de ce qu'ils considéraient comme le code source.

Deux perspectives différentes pourraient être mises de l'avant pour rendre davantage visibles la diversité des contributions au développement des logiciels libres. La première, déjà proposée par quelques auteures féministes s'étant attardées aux pratiques des logiciels libres, consiste à valoriser d'autres activités que celles directement reliées au code source. C'est ainsi que Lin écrit que : « To make FLOSS successful, we need not only Richard Stallman or Linus Torvalds, but also a great amount of volunteers reporting and fixing bugs, writing documentation, and more importantly, teaching users how to use the program » (Lin, 2006a, p. 1149). Haralanova renchérit sur ces propos de Lin et d'autres femmes impliquées dans le mouvement du logiciel libre, en écrivant que : « le codage n'est ni l'unique ni la plus importante activité du processus d'innovation du libre » (Haralanova, 2010, p. 43). Haralanova propose ainsi dans son mémoire de maîtrise une exploration des différentes activités, liées ou non à la « contribution au code source ».

La perspective développée dans notre thèse nous amène à des conclusions similaires et étroitement liées sur le plan politique, mais symétriques sur le plan conceptuel. Plutôt que de nous intéresser aux activités qui ne sont pas liées au « codage » et à la « contribution au code source », il s'agit plutôt de nous intéresser à la manière dont sont définis le « codage » et la « contribution au code source » et, ultimement, de remettre en question ces définitions[192]. D'un point de vue féministe (ou critique), il serait également intéressant de reprendre la définition du code source donnée par la GPL comme la « forme privilégiée du travail pour réaliser des modifications », en nous questionnant sur ces formes qui sont privilégiées, et en nous demandant qui les privilégie. Ce flou entourant la notion de code source ouvre par conséquent la voie à l'étude d'autres formes d'interaction et de reconfiguration qui s'apparentent à l'écriture du code source. C'est dans cette perspective que nous suggérons à la section 7.2.2 (*Code source et autres formes d'écrits numériques*) d'aborder les similitudes entre le code source et d'autres formes d'écriture numérique.

---

192 Ainsi, cette question, que nous avions posée lors d'un panel pour la conférence 4S (voir le chapitre 1), mériterait d'être approfondie : « Quelles sont les formes de travail avec les ordinateurs qui sont considérées comme du codage et lesquelles ne le sont pas ? ».

7.1.2 La performativité du code source

Comment la fabrication et la stabilisation du code source s'articule-t-elle à sa force « législative », ou performative ?

La question de la performativité a traversé l'ensemble de notre travail. D'une part, notre approche théorique, décrite au chapitre 2, s'inscrit de façon générale dans le « programme performatif » visant à contrer les perspectives essentialistes, en s'intéressant plutôt au travail de performation nécessaire à la fabrication et la stabilisation des catégories et des assemblages. D'autre part, nous nous sommes plus précisément attaché à analyser la performativité du code source. Dans notre problématique et notre cadre théorique, nous avons cité les travaux d'Adrian Mackenzie (2005; 2006) qui s'est attardé à la question de la performativité du code informatique. Rappelons que dans son analyse, Mackenzie s'appuie surtout sur le développement que fait Butler de la performativité, qui considère qu'un énoncé ne peut effectivement être performatif que s'il s'appuie sur des discours et des pratiques antérieurs. Pour Butler, si un énoncé performatif réussit, c'est parce qu'il « *accumule la force de l'autorité à travers la répétition ou la citation d'un ensemble de pratiques antérieures qui font autorité* » (Butler, 2004, p. 80)[193]. Dans cette perspective, la programmation est appréhendée pour Mackenzie comme une pratique continuelle de citation du code. La performativité du code dépendrait de la répétition de ce qu'il nomme l'*authorizing context,* terme que nous avons traduit par celui de « contexte d'autorisation », qui serait défini comme l'ensemble des conventions et des pratiques dont la répétition donne une force performative. Ainsi, dans son analyse du cas de Linux, la performativité du code reposerait sur la réitération d'un ensemble de pratiques d'administration informatique, de programmation et de design logiciel quelques fois désignées par le terme de « philosophie Unix » (Mackenzie, 2005, p. 84).

Dans le cadre de notre thèse, nous nous sommes cependant davantage appuyés sur les travaux récents concernant la performativité des artefacts, notamment ceux réalisés dans la perspective d'une anthropologie de l'écriture (Fraenkel, 2006; Denis et Pontille, 2010a; Denis et Pontille, 2010b). Rappelons que ces derniers travaux s'attardent à montrer que la performativité des artefacts écrits repose sur un travail incessant de stabilisation et sur leur

193 Les italiques sont dans le texte.

insertion réussie dans des agencements sociomatériels. Denis et Pontille (2010b), dans leur étude sur la signalétique du métro de Paris, montrent par exemple le rôle de la spatialité et du travail de maintenance dans la performativité des panneaux du métro de Paris. Pour qu'un panneau performe l'effet désiré d'orienter les voyageurs dans le métro, il doit répondre à certaines normes graphiques et être situé dans des endroits significatifs dans le métro, autant de propriétés qui dépendent d'un important travail de fabrication et de maintenance. Dans une perspective similaire, Fraenkel soutient pour sa part que la performativité de l'acte écrit réside notamment dans l'agencement de cet acte dans une chaîne d'écriture plus large. Ainsi, dans un article où elle critique la conception de l'écrit dans la théorie de la performativité d'Austin (1975), Fraenkel (2006) affirme que la performativité du testament repose en bonne partie sur un travail visant à donner une cohérence juridique à l'acte du testament, en le mettant en forme et en apposant les sceaux et les signatures appropriés. Il s'agit donc dans ces travaux, non pas tant de s'intéresser à l'*effet* performatif des artefacts, mais plutôt à ce que Bourdieu (1975) appelait les « conditions de félicités » de cette performativité, mais d'une manière qui prenne en compte la dimension matérielle de la performativité, de même que le travail de performation nécessaire à celle-ci. C'est avant tout dans cette perspective que nous avons cherché à appréhender la performativité du code source.

En effet, si la particularité du code source de *faire agir une machine* semble lui donner une performativité « intrinsèque »[194], le questionnement devient cependant plus complexe lorsqu'il s'agit de demander : *quel code fait agir quelle machine ?* En effet, dans les projets étudiés, le code source apparaît comme un artefact, ou un ensemble d'artefacts dispersés, qui prend la forme de morceaux de code source, de rustines, de plugins, de « code snippets », ou encore simplement, de prototypes de code source partagés par courriel ou par IRC, voire même en personne. Ces artefacts circulent sur une nébuleuse de sites web, blogues, wikis et autres espaces électroniques de discussion, nébuleuse décrite dans le cas de SPIP comme

194 Cet aspect performatif du code source ressortait bien dans cette entrevue que nous avons citée au chapitre 4 (p. 159) et lors de laquelle notre interlocuteur comparait le code source à des incantations magiques : « tu prononces des mots et ça produit un effet. C'est pas ça la magie? » (spip01). Cette description du code source résonne de façon remarquable avec le titre de l'ouvrage de Austin sur la performativité, *How to do things with words* (Austin, 1975), ou en français *Quand dire c'est faire* (Austin, 1991). D'une certaine façon, la performativité du code informatique n'est pas à démontrer puisque, par définition, le code informatique a pour rôle de faire agir un ordinateur.

étant la « galaxie SPIP » (voir chapitre 3, p. 102). Ainsi, bien que le premier niveau de la performativité consisterait à convenir que la propriété même du code source est de faire agir (une machine), un second niveau devrait nous amener à penser que la capacité à agir d'un morceau de code, sa force performative, est également liée à son emplacement dans le réseau sociotechnique. Un morceau de code source sur lequel un individu travaille seul dans son sous-sol, et qui ne sera jamais utilisé autrement, n'aura pas la même performativité qu'un autre morceau de code source utilisé, par exemple, dans l'exécution du site Daily Motion. Cet aspect se compare par exemple avec la signalétique, étudiée par Denis et Pontille (2010a; 2010b). Un panneau « signalétique » fabriqué chez soi et laissé à l'abandon dans un débarras n'aura, de toute évidence, pas la même performativité (s'il en a tout court) qu'un autre panneau signalétique, celui-ci fabriqué par une organisation autorisée, répondant à certains standards graphiques, placé dans des endroits stratégiques dans le métro, et régulièrement entretenu. En d'autres termes, la performativité d'un artefact ne dépend pas uniquement de ses propriétés intrinsèques, mais dépend également, et peut-être surtout, de son insertion dans un réseau sociotechnique élargi.

Cette aspect a été surtout approfondi dans le chapitre 6. Nous avons d'abord forgé le concept de « code source autorisé » pour désigner le code source disponible en téléchargement, ce concept s'inspirant de celui de « langage autorisé » mis de l'avant par Bourdieu (1975) pour insister sur les conditions sociales de la performativité du langage. Contrairement à l'ensemble du code source qui circule autour du projet, la version disponible sur la page de téléchargement a ceci de particulier qu'elle constitue uniquement cette partie du code source qui a été décrétée à un moment donné, et par certaines personnes, comme étant une version *stable* et autorisée à agir. Nous inspirant de l'anthropologie de l'écriture de Fraenkel, notre analyse s'est ensuite concentrée sur l'acte informatique du commit, qui consiste à valider une modification au code source et à insérer cette modification dans le code source éventuellement disponible au téléchargement. À bien des égards, le commit pourrait être comparé à l'acte de la signature, amplement discuté par Fraenkel (2006; 2007; 2008), en ce sens qu'il s'agit d'un acte qui participe à la validation du code source éventuellement

disponible en téléchargement[195]. Notre analyse a également montré que cet acte, probablement du fait de son rôle important en regard de la performativité du code source, est étroitement lié à certains droits et autorisations, ainsi qu'à une écologie de la visibilité qui rendent certaines formes de contributions au code plus visibles que d'autres. En bref, dans le chapitre 6, nous avons tenté de montrer que la performativité du code source dépend d'un important travail de stabilisation, lui-même étroitement articulé à différentes dynamiques d'autorité et de visibilité.

Nous avons également mis de l'avant au chapitre 5 la nécessité, pour qu'un morceau de code source soit performatif, d'être inséré dans un ensemble plus large, de répondre à certaines règles et conventions d'écriture, ou de codage. Ces règles sont parfois « fortes », en ce sens qu'elles sont syntaxiquement inscrites dans le langage ou dans le design des interfaces : le non-respect de ces règles empêche l'exécution du programme informatique. D'autres règles sont plus « faibles », en ce sens que le respect ou non de celles-ci n'aura pas d'impact sur l'exécution du logiciel. Néanmoins, ces conventions dites faibles sont avant tout justifiées de la part des acteurs par le désir d'une certaine unité au sein du code source. Leur caractère prescriptif est renforcé par les différents dispositifs d'autorité et d'autorisation mis en place par les acteurs. Ainsi, une contribution au code source – par exemple une rustine – pourra être refusée si ces conventions faibles ne sont pas respectées. Ce lien entre les conventions d'écriture et la performativité du code source rejoint l'analyse de Fraenkel qui, citant Latour (2004), note à propos de l'acte juridique, « qu'il existe une relation directe entre la force exemplaire de l'acte juridique et le réglage précis de ses formes textuelles, graphiques, matérielles » (Fraenkel, 2006). Dans le cas qui nous intéresse, les conventions fortes et les conventions faibles participent à la performativité d'un morceau de code source, car elles permettent d'en faciliter l'insertion dans le code source autorisé. Comme nous l'avons

195 Comme nous l'avons décrit au chapitre 6 (p. 213 et p. 247), cette comparaison avec l'acte de signature a cependant ses limites. D'une part, la définition du code source n'étant pas stable, il ne peut dans tous les cas être assimilé à un écrit. D'autre part, et de façon plus importante, alors que la signature confère à l'acte juridique une valeur d'authenticité, l'acte du commit a plutôt comme conséquence pragmatique d'appliquer une modification précise au code source déjà reconnu comme autorisé. La comparaison reste cependant pertinente en ceci que la signature, comme le commit, sont des actes précis qui ont comme conséquence pragmatique de participer à l'insertion d'un artefact – l'acte juridique dans un cas, la modification au code source dans un autre – dans un réseau sociotechnique plus large.

mentionné en conclusion du chapitre 5, les normes d'écriture, en particulier dans le contexte du logiciel libre, participent également à l'« immuabilité mobile » (Latour, 1985; Denis et Pontille, 2010a, p. 111) d'un morceau de code source. En effet, la performativité d'un morceau de code source dépend de sa capacité à être reproduit et inséré dans différents réseaux. Dans cette perspective, la compatibilité des normes et des conventions de différents projets participent à la mobilité des morceaux de code source. La conformité d'un morceau de code source aux normes et conventions permet donc au code source d'être « mobile » entre ces projets, d'être reproduit d'un réseau de code source à un autre et ce, sans être modifié à chaque fois, c'est-à-dire, en restant « immuable ».

C'est sur ce dernier point que notre analyse de la performativité du code source rejoint l'analyse que fait Adrien Mackenzie (2005) de la performativité. En effet, de la même manière que Mackenzie considère que la performativité du code dépend de la réitération d'un ensemble de pratiques et de normes, notre analyse met de l'avant que la performativité du code source dépend en bonne partie du respect, et donc de la réitération, de certaines normes d'écriture et pratiques de codage. De l'analyse culturelle proposée par Mackenzie, nos descriptions nous amènent à analyser de façon plus précise et pragmatique le rôle de ces conventions et normes d'écriture dans les pratiques du code source.

Mentionnons en terminant que notre analyse de la performativité du code source pourrait être davantage approfondie. L'une des principales limites de notre analyse est en effet que nous nous sommes limités à analyser le travail de stabilisation du code source se situant en amont de son autorisation à figurer sur la page de téléchargement. Hors, l'analyse de ce travail de stabilisation, bien que nécessaire, ne suffit pas à expliquer la performativité du code source. En effet, une fois le code source stabilisé et disponible sur la page de téléchargement, la performativité du code source ne va pas de soi, car il y a encore tout un travail d'installation, d'articulation, de réglage et et de maintenance pour assurer la vitalité du code source. Cette analyse de ces opérations qui se situent à la charnière du travail de conception et de l'usage, constitue une des voies qu'il nous semble intéressant d'explorer dans le futur.

7.1.3 Valeurs et design du code source

De quelles manières certaines valeurs sont-elles inscrites dans la fabrication et le design du code source ?

L'un des aspects importants qui ressort de notre analyse concerne, d'une part, l'inscription des valeurs de chacun des projets dans le code source et, d'autre part, l'organisation du code source de même que le déploiement des règles et des autorisations qui participent à sa fabrication. Rappelons que la notion d'inscription est utilisée dans la théorie de l'acteur-réseau pour exprimer comment l'objet technique définit un cadre qui délimite l'action des acteurs et de l'environnement. Plusieurs auteurs se sont inspirés de la notion d'inscription pour saisir l'articulation de certaines valeurs au design de l'objet technique. Ainsi, Serge Proulx énumère plusieurs niveaux d'analyse dont l'un serait « L'inscription de dimensions politique et morale dans le design de l'objet technique [...] ». Il s'agirait, selon Proulx, de montrer comment la conception et l'usage des objets techniques sont porteurs de représentations et de valeurs politiques et morales ou encore, comment certains rapports sociaux sont « contenus » dans le design de l'objet technique. Dans une perspective similaire, nous avons également mentionné (chapitre 3, p. 89) l'ethnographie des infrastructures de Star qui consiste notamment à faire ressortir la manière dont certaines valeurs et principes éthiques sont inscrits dans les profondeurs de l'environnement informationnel (Star, 1999, p. 379).

Cet aspect d'inscription des valeurs dans le design du code source ressort particulièrement de la comparaison des deux projets étudiés. Dans la présentation des deux projets (chapitre 3), nous avons mentionné les valeurs distinctes de chacun des projets – nettement anarchisantes et non-commerciales dans le cas de SPIP, plus autoritaires et orientées vers des questions commerciales dans le cas de symfony. Nous avons également noté que ces valeurs s'exprimaient par exemple dans la manière dont les rencontres publiques étaient organisées. Alors que les rencontres de symfony sont commanditées par des entreprises informatiques telles que Microsoft ou Yahoo, avec un coût d'inscription assez élevé, celles organisées dans le cadre de SPIP sont gratuites et organisées sur une base beaucoup plus collégiale. Nous avions également noté que ces différences en termes de valeurs s'exprimaient également dans le type de licence privilégié par chacun des projets. Alors que la licence publique générale GNU (GPL) est la licence privilégiée pour SPIP, son utilisation est interdite dans le cas de symfony. Cette distinction s'explique par le désir ou non de permettre la réutilisation du code

source dans le cadre du développement d'applications propriétaires (la GPL interdit une telle réutilisation, tandis que la licence MIT, privilégiée par symfony, le permet). Cette première constatation constitue déjà un exemple frappant d'une certaine inscription des valeurs de chacun des projets dans leur code source et, plus précisément, dans les artefacts légaux qui régulent la fabrication collaborative et le partage du code source. Dans l'optique des valeurs plus « autoritaires » de symfony, les normes d'écriture y sont par ailleurs beaucoup plus rigoureuses et ont été établies a priori, dès le début du projet. Dans SPIP, au contraire, le projet a débuté sans nécessairement avoir de normes d'écriture et, bien que certaines règles aient été mises en place, le code source de SPIP reste en général assez faiblement uniformisé. L'inscription des valeurs du projet est également notable dans le choix de la syntaxe pour le langage de squelettes dans SPIP ou du format utilisé pour la configuration des formulaires dans symfony (voir section 5.1). Mentionnons également qu'à la différence de SPIP, l'écriture du code source de symfony repose davantage sur des « bonnes pratiques » et des standards reconnus.

Dans le chapitre 6, nous avons noté que l'octroi des droits de commiter dans SPIP est explicitement conditionnel à l'adhésion aux valeurs du projet, stipulées dans la « Charte de la Zone ». Sans que la référence à des valeurs soit faite de manière aussi explicite, l'octroi des droits de commiter dans symfony est quant à lui conditionnel à l'utilisation de licences qui permettent, comme nous l'avons mentionné plus tôt, la réutilisation du code source dans des logiciels propriétaires. De manière plus générale, il nous semble que les valeurs de chacun des projets s'articulent également avec les différentes mises en œuvre des autorisations de commit dans chacun des projets. Nous avons ainsi décrit comment les autorisations de commit sont beaucoup plus restrictives dans le cas de symfony que dans celui de SPIP. Ces différences sont particulièrement marquées au niveau des plugins. Dans symfony, un système complexe d'autorisation est mis en œuvre au niveau des plugins. Dans SPIP, au contraire, seules deux conditions sont nécessaires à l'obtention de ces droits : en faire la demande sur la liste et adhérer à la Charte. Mentionnons finalement que les deux projets se distinguent également dans leurs valeurs par rapport à l'importance accordée à l'anglais, dans le cas de symfony, et au français, dans le cas de SPIP.

Nous voyons ici que le code source n'est pas quelque chose d'inerte, mais est au contraire infusé d'autorité, de valeurs et de relations de pouvoir. Chacune des parties du code source est

articulée à différentes autorisations et autorités. Ces valeurs distinctes qui ressortent dans SPIP et symfony ouvrent la porte sur un questionnement plus large concernant les politiques du logiciel libre : au-delà de la valeur de la « liberté » mise de l'avant par les militants du logiciel libre (ou celle de l'« ouverture » par les militants de l'*open source*), quelles autres valeurs spécifiques sont articulées, ou inscrites, dans le code source de logiciels libres particuliers ?

7.1.4 Le code source comme interface dans les reconfigurations d'Internet

Comment le code source agit-il dans les reconfigurations d'Internet ?

La réponse à cette dernière question de recherche nous amène au cœur de notre thèse. La proposition que nous souhaitons défendre ici est que le code source agit comme *interface* dans les reconfigurations continuelles d'Internet. Cette conceptualisation de la capacité d'agir du code source a émergé progressivement durant notre étude, et prend appui sur notre exploration théorique et notre travail empirique.

Sur le plan empirique, la conception du code source comme interface a émergé à la suite d'une de nos premières entrevues avec un acteur de symfony qui mettait de l'avant l'idée d'une ergonomie du code source (chapitre 5, p. 206). Rappelons que pour cet acteur, la manière dont le code source est conçu, écrit et organisé affectera sa capacité d'utilisation. En particulier, le choix d'un format, ou d'un langage de programmation plutôt qu'un autre (le YAML plutôt que le PHP, dans le cas que nous avons analysé), aura des conséquences sur la facilité d' « utilisation » du code source. Dans cette perspective où l'on peut parler d'une « ergonomie du code », nous disait cet acteur, il convient dès lors de considérer le code source comme une interface, au même titre que les boutons et les fenêtres dans d'autres logiciels. Citons de nouveau cet extrait de l'entrevue :

> Quand je suis un graphiste, mon interface, c'est Photoshop, avec ses boutons, ses fenêtres, etc. Quand je suis un développeur, mon interface, c'est le code. C'est avec ça que j'interagis avec la matière que je construis, étant un programme. Alors que le graphiste, il va construire une image (sf03).

Le concept d'interface est également présent dans le vocabulaire des acteurs par leur référence aux API – *Application Programming Interfaces* ou interfaces de programmation applicative. Les API sont en quelque sorte des « prises » dont les acteurs peuvent se saisir

pour manipuler un morceau de code source sans nécessairement avoir à comprendre ce qui se trouve *à l'intérieur.* De plus, les API permettent de relier entre eux les différents morceaux du code source. Nous l'avons mentionné, cette conception de l'interface est particulièrement notable concernant les squelettes de SPIP et les formats de configuration YAML dont le statut de code source, ou de langage de programmation, n'est pas tout à fait clair. Si certains considèrent qu'il s'agit de code source, une actrice expliquait quant à elle que les fichiers YAML, c'est ce qui « configure le code source, mais ça ne fait pas partie du code lui-même » (sf05). C'est dans ce sens que nous pouvons considérer les squelettes, ou les fichiers YAML, comme des *interfaces* pour configurer le code source. De manière générale, cependant, le nom d'une fonction ou d'une procédure au sein du code source peut être appréhendé comme une interface, dans ce sens que c'est par l'intermédiaire de ces interfaces qu'il est possible de manipuler, ou de reconfigurer telle ou telle partie du code source. Dans cette perspective, l'ensemble du code source peut être appréhendé comme la composition d'une multiplicité d'interfaces.

Cependant, si nous nous appuyons en partie sur les propos des acteurs pour considérer le code source à la manière d'une interface, notre proposition est avant tout d'ordre théorique. En effet, considérer le code source comme une interface semble contre-intuitif pour certains acteurs. Contrairement à l'acteur cité précédemment (sf03), qui considérait le code comme une interface, un autre acteur considère que symfony n'aurait pas d'interface, à moins de proposer une conception étendue de cette notion :

> *Q - Est-ce que tu dirais que symfony a une interface ?*
> Hum... [pense] On peut peut-être le dire en étendant la notion d'interface. En fait, l'interface du produit fini, c'est le site que j'ai conçu. Mais il y a pas d'interfaces à symfony. En soi, symfony, ce n'est rien, c'est juste des briques, qui vont me permettre de construire mon produit. Donc, si on veut parler d'interface de symfony, en tant que logiciel d'aide à la construction d'autres logiciels, ça pourrait être la documentation éventuellement. C'est un peu tiré par les cheveux je crois (sf06).

C'est dans cette perspective que nous souhaitons articuler la conception du code source comme interface à nos réflexions théoriques. En effet, bien que cette conception de l'interface semble pour cet acteur « un peu tiré[e] par les cheveux », elle semble toutefois rejoindre l'approche de Suchman (2007). Rappelons que l'approche de Suchman fait contre-pied à cette conception de l'interaction humain-machine vue comme une « conversation » en temps réel

entre l'humain et la machine, pris comme entités autonomes, à travers une interface transparente et stable. Suchman propose au contraire de poser le regard sur des dynamiques de reconfigurations mutuelles et permanentes des relations entre les humains et les machines. Dans cette perspective performative, Suchman refuse de poser l'interface comme une frontière « fixe » et a priori entre l'humain et la machine. L'interface est plutôt une relation entre humains et machines, et entre humains entre eux, sinon toujours changeante, du moins continuellement performée. Dans une telle conception, la « conversation » entre l'humain et la machine n'est pas tant un moment d'échange de messages, mais devrait plutôt être appréhendée comme une production continuelle et contingente d'un monde sociomatériel partagé (Suchman, 2007, p. 23). L'interface est donc pensée en situation, comme un élément dans un « réseau étendu de relations arbitrairement découpées » (Suchman, 2007, p. 45). Dans cette perspective, sur le plan empirique, si la documentation peut être appréhendée comme une interface, il pourrait être possible d'étendre cette notion à l'ensemble des dispositifs sur lesquels les acteurs s'appuient pour fabriquer, configurer ou reconfigurer les applications web.

Cette perspective consistant à considérer le code source comme une interface trouve un certain écho dans un article publié par Vaucelle et Hudrisier (2010) dans la revue *TIC et société*. Dans cet article, les auteurs proposent de considérer la manière dont l'utilisation des langages à balises, tels que html ou XML, ou encore des cadres d'application (*framework*) « réorganisent l'aménagement d'une interactivité ». Remettant en question une conception dominante de l'interactivité qui ne découlerait que du « temps réel », les auteurs citent Simondon et soutiennent qu'interagir avec une machine, « c'est toujours interagir avec de la pensée humaine que l'on parle d'un simple marteau ou d'une application informatique sophistiquée » (Vaucelle et Hudrisier, 2010, p. 56). Ils écrivent également que « quand nous communiquons avec un logiciel ou une machine, nous sommes également en rapport à la créativité de celui (ou dans le cas le plus fréquent, de ceux) qui les a (ont) mis au point et inventé » (Vaucelle et Hudrisier, 2010, p. 57).

Pour Vaucelle et Hudrisier (2010), si dans des logiciels comme *Word*, la situation est proche de la « *doxa* interactive », le cas des téléphones mobiles est moins asymétrique. L'utilisateur a en effet davantage de possibilités pour redéfinir les comportements de l'interface et pour l'organiser au plus près de ses besoins et de ses préférences d'interaction. Selon les auteurs,

les langages de balises et les *frameworks* exploitent cependant une autre conception de l'interactivité, par ailleurs très innovante (selon eux), en agissant à la manière de coquilles où l'utilisateur pourra entrer ses propres couches d'information.

Mentionnons qu'en conclusion de leur article, Vaucelle et Hudrisier (2010) notent que le futur du « web sémantique » ne s'appuie pas tant sur les perfectionnements du web imaginés par les chercheurs en informatique, mais « qu'il s'appuiera certainement sur l'appropriation sociale par des individus aujourd'hui enfants ou adolescents qui manipuleront des objets médiatiques interactifs mutants, mais éminemment convergents » (Vaucelle et Hudrisier, 2010, p. 65). Dans cette perspective, selon les auteurs, les langages structurés et les cadres d'application occupent une place de premier plan par la manière dont ils permettent d'intégrer différents niveaux d'interaction aujourd'hui encore très hétérogènes. Pour ces auteurs, le devenir de l'interactivité dépendrait non seulement des développements technologiques, mais également des mutations langagières qui sont encore largement imprévisibles. On voit bien comment cette perspective résonne avec celle de Suchman qui propose d'appréhender les interactions humain-machine comme des reconfigurations mutuelles et continuelles des relations entre humains et machines. Elle résonne également avec la proposition de Suchman de recourir aux métaphores de la lecture et de l'écriture pour analyser les reconfigurations mutuelles, en insistant sur le fait qu'il s'agit d'autres formes de lecture et d'écriture que celles auxquelles nous sommes habitués (Suchman, 2007a, p. 33)[196].

Découpage et interfaces

Il nous semble cependant important d'aller un peu plus loin en suivant la manière dont Suchman conçoit le caractère politique des interfaces. L'interface n'est en effet plus une entité établie a priori : elle désigne plutôt certaines formes de « découpages » qui se produisent dans toutes les pratiques sociomatérielles :

---

196 Dans une perspective beaucoup moins théorique, mentionnons cet article du *New York Times* à propos de l'essor de la formation populaire concernant les rudiments des langages de programmation d'Internet (Wortham, 2012). Les personnes qui suivent ces cours, des « non-techies », le font parce qu'il est de plus en plus nécessaire de pouvoir configurer des blogues ou des sites Internet dans le cadre de leur travail. « Inasmuch as you need to know how to read English, you need to have some understanding of the code that builds the Web », soulignait l'une des personnes rencontrées en entrevue.

> Brought back into the world of technology design, this intimate co-constitution of configured materialities with configuring agencies clearly implies a very different understanding of the 'human-machine interface'. Read in association with the empirical investigations of complex sites described above, 'the interface' on the one hand becomes the name for a category of contingently enacted 'cuts' occurring always *within* sociomaterial practices, that effect 'persons' and 'machines' as distinct entities [...] (Suchman, 2006, p. 6)

Or, nous dit Suchman, ces découpages et ces interfaces ne sont pas neutres et donnés naturellement. Ils sont plutôt historiquement construits et ont des conséquences sociales et matérielles. Ces configurations sont porteuses de certaines figures de l'humain et de la machine. Une possibilité critique consiste à comprendre la façon dont les humains et les machines sont figurés dans ces pratiques et la façon dont des figures alternatives – des reconfigurations – peuvent potentiellement remettre en question les régimes actuels de production scientifique et technologique (Suchman, 2007, p. 228)[197].

Cette différentes figures de l'humain – et plus spécifiquement, de l'usager – et de la machine, de même que leurs possibilités de reconfiguration, ont été particulièrement explorées au chapitre 5. Dans les deux projets, nous avons constaté différentes formes d'interfaces pour (re)configurer l'application, ou le reste du code source. Dans SPIP, on constate par exemple une différence notable entre les plugins, développés en langage PHP, et les squelettes, développés dans un langage « maison ». Le fait que ce langage soit fait « maison » et qui plus est, en français, participe sans doute à la faible diffusion de SPIP hors de la France. Plusieurs acteurs que nous avons rencontrés en entrevues ont manifesté une grande appréciation de cette interface « de squelettes », à la fois parce qu'elle était en français – seule langue « humaine » véritablement maîtrisée par plusieurs des participants à SPIP –, mais aussi parce

197 Comme nous l'avons décrit au chapitre 2 (section 2.2.3 p. 65), rappelons que le concept de reconfiguration est au centre de l'approche développée par Suchman (2007). Bien que ce concept ne soit pas explicitement défini par l'auteure, celle-ci fait principalement référence au travail de Haraway (1991) pour qui les langages et les technologies sont figuratifs dans ce sens qu'ils invoquent toujours certaines associations de sens et de pratiques. Configurer, dans cette perspective, signifie en quelque sorte d'incorporer certaines figures – des associations de sens et de pratiques – au sein de l'assemblage sociomatériel. Suchman fait d'ailleurs référence au texte de Woolgar (1991) sur la « configuration de l'usager » qui analyse la manière dont le design d'un dispositif technique (un micro-ordinateur) implique une construction d'une figure de l'usager qui sera éventuellement incorporée dans le dispositif technique qui prescrira ensuite le cours de l'action de l'usager réel (d'où l'idée d'une configuration de l'usager). Suivant cette approche, reconfigurer implique donc de configurer les assemblages sociomatériels à partir de figures alternatives.

que l'interface n'incorpore pas une culture trop « informatique ». Bref, nous posons ici l'hypothèse que l'interface de SPIP, tout comme celle de symfony, opère des découpages concernant le « public » visé : international (lire : de langue anglaise) et issu d'une culture informatique pour symfony, plutôt Français, mais rejoignant un plus large spectre social dans le cadre de SPIP. Le design des interfaces qui composent le code source, voire la manière même dont le code source est défini, a des implications politiques en ce sens qu'il favorise la participation de certains acteurs plutôt que d'autres.

En d'autres termes, la manière dont l'interface de telle ou telle partie du code source est conçue implique un certain « découpage » de l'une des boîtes noires avec laquelle un utilisateur interagira. S'il y a une boîte noire, la noirceur effective de celle-ci de même que ses contours sont loin d'être définis d'avance, de façon essentialiste. *Le code source n'est pas par définition une boîte noire.* Plus spécifiquement, nous avons tenté à travers notre analyse de faire ressortir que le code source, loin d'être *une* boîte noire, consiste plutôt en une *multitude* de boîtes – des morceaux de code encapsulés et interreliés les uns aux autres. Si chacun de ces morceaux de code source encapsulés peut être appréhendé comme une boîte, au sens de brique, ils peuvent également être pensés comme des boîtes noires, au sens de la théorie de l'acteur-réseau. En d'autres termes, si les acteurs que nous avons rencontrés sont pour la plupart des concepteurs de certaines parties du code source, ils doivent également être considérés comme des usagers d'autres parties du code source qu'ils appréhendent comme autant de boîtes noires dont l'interface masque la complexité interne de celles-ci.

Au final, appréhender le code source comme une interface renvoie à l'idée que cet artefact n'est pas seulement le noyau d'une technologie, mais qu'il est aussi un artefact à travers duquel les acteurs interagissent *entre eux,* entrent en relation, pour configurer, ou reconfigurer les dispositifs socionumériques d'Internet. La métaphore de l'interface met également de l'avant l'idée que la relation avec le code source peut en être une d'usage, et non pas seulement de conception[198]. L'erreur qu'il faut éviter est donc d'associer, de manière essentialiste, « code source » et « boîte noire ». Dans cette perspective, le code source informatique ne doit pas être considéré uniquement comme un artefact dans la conception

198 Nous pourrions en fait parler d'un continuum entre activités de conception, de configuration et d'usage dans le cas des projets étudiés.

d'un dispositif technique, mais également (et peut-être avant tout) comme un espace de reconfigurations continuelles et mutuelles des humains et des machines.

7.2 Pistes futures de recherche

7.2.1 Cultures de « codage » et valeurs dans le design du code source

Nous avons abordé la question des valeurs dans le design du code source dans le cadre de notre analyse, mais celle-ci mériterait à notre avis d'être approfondie. À cet effet, mentionnons que nous avons participé, durant la durée de notre doctorat, à un réseau de recherche sur la question des « valeurs dans le design »[199]. Bien que ce regroupement académique n'ait pas fait émerger jusqu'à présent une théorisation englobante, plusieurs travaux réalisés dans cette perspective pourraient être recoupés avec les nôtres. Nous avons déjà mentionné les travaux de Star et Bowker, qui figurent parmi les quelques chercheurs ayant initié ce réseau, et pour qui l'un des aspects cruciaux de l'analyse des infrastructures est la manière dont certaines valeurs sont inscrites au cœur des projets (Bowker et Star, 2000b; Star, 1999).

Dans une perspective plus empirique, il pourrait également être intéressant de s'attarder aux cas de développement de logiciels assez distants du domaine de l'informatique professionnelle. À cet effet, mentionnons une entrevue que nous avons réalisée en octobre 2011 avec un doctorant en sciences de l'environnement de l'UQAM dont le travail de recherche nécessite de réaliser des simulations météorologiques, à l'aide d'un logiciel « maison » développé au fil des années par un réseau de chercheurs interuniversitaires en climatologie et météorologie. Pour réaliser ses propres simulations, notre interlocuteur devait lui-même modifier le logiciel pour l'adapter à ses besoins, puis partager le code source avec les autres chercheurs. Les descriptions que faisait notre interlocuteur de l'interaction avec le code source montraient des différences importantes avec celles que nous avons faites dans le cadre de cette thèse. Ayant lui-même une certaine expérience des pratiques décrites dans notre étude, telle que l'usage de SVN ou Git, notre interlocuteur mettait de l'avant les difficultés de faire collaborer une équipe étendue de chercheurs à travers le code source. Ces

199 Voir par exemple le dernier atelier qui a réuni les membres de ce réseau : <http://sites.google.com/site/vid2k10workshop/home> (consulté le 29 mars 2012).

descriptions ouvrent la voie, non seulement à l'articulation des valeurs dans le design du code source et des dispositifs de fabrication collective, mais également à la mise en relation de notre étude sur le code source avec celles concernant les infrastructures d'information scientifique (Bowker et al., 2010; Millerand et Baker, 2010). En effet, le code source dans ce cas-ci pourrait être appréhendé à la fois comme l'infrastructure de la collaboration, mais également en tant qu'objet de la connaissance, celui-ci étant fabriqué par les scientifiques eux-mêmes, dans l'objectif de parfaire les simulations informatiques des phénomènes météorologique et climatiques. Reprenant l'approche que nous avons développée dans notre thèse, il s'agirait ainsi de s'attarder à la manière dont ces chercheurs collaborent dans la fabrication et l'entretien du code source et symétriquement, de voir comment le code source lui-même participe à la « structuration » de cette collaboration.

7.2.2 Code source et autres formes d'écrits numériques

Une autre piste de recherche à explorer consisterait à comparer le code source avec d'autres formes d'écrits numériques. De la même manière, il s'agirait de comparer les pratiques d'écriture du code source à d'autres pratiques d'écriture numérique contemporaine. Ces comparaisons sont déjà implicites dans plusieurs études qui se sont attardées aux dynamiques du web 2.0. Ces études notent en effet souvent la manière dont les pratiques du logiciel libre ont été précurseures d'autres pratiques, comme le développement de Wikipédia ou encore les sites de réseaux sociaux. C'est dans cette perspective que Cardon (2005) propose par exemple le terme d'innovation ascendante, ou d'innovation par l'usage, en notant que le mouvement du logiciel libre a donné un cadre normatif et organisationnel à l'innovation ascendante.

Si ces études sont intéressantes, les liens qu'elles établissent entre les logiciels libres et d'autres dynamiques d'innovation ascendante restent surtout au niveau des *pratiques* plutôt que de l'analyse du *média* lui-même. Ainsi, dans l'analyse de l'innovation ascendante que fait Cardon (2005) se situe surtout au niveau des discours et des modes d'organisation des communautés de logiciels libres ou de Wikipédia. L'artefact qui est l'objet de la pratique, le code source dans le cadre des logiciels libres ou le texte en format wiki dans le cadre de Wikipédia, est cependant laissé de côté de cette analyse. Par exemple, comment la forme que prend le code source d'un logiciel ou celle que prend le texte d'un wiki affecte-t-elle les pratiques collaboratives des acteurs qui interagissent avec ces artefacts ? Quelles similitudes

existe-il entre ces formes textuelles ? Nous proposons ici deux bases de comparaison entre le code source et d'autres artefacts numériques, qui pourraient être explorées dans de futures recherches.

Code source et wiki

Les liens historiques et pragmatiques entre le wiki et le code source nous semblent une avenue intéressante à analyser. D'une part, sur la plan historique, nous avons déjà mentionné au chapitre 5 les liens entre les origines du wiki et certaines « bonnes pratiques » que l'on retrouve notamment dans le projet symfony. Il est par exemple remarquable que l'inventeur du premier wiki, Ward Cunningham, soit également l'un des signataires du manifeste *Agile*, qui se veut l'un des premiers documents faisant la promotion du développement de logiciels collectifs. Remarquable également que le premier wiki ait été mis en place (et soit toujours utilisé) par la communauté des « Design Patterns », ces motifs de conception qui occupent un rôle normatif important dans le projet symfony.

Sur le plan pragmatique, mentionnons tout d'abord cette description que fait Anne Goldenberg en avant-propos de sa thèse de doctorat, concernant son rapport aux wikis :

> Avec la syntaxe minimaliste de wikis que j'abordais, syntaxe que je rapprochais alors à un début de programmation, je jouais avec la mise en forme et l'infrastructure de divers contenus [...] Et lorsque j'éditais un wiki, j'avais la sensation d'avoir sous les doigts un matériau maniable comme une pâte à modeler numérique (Goldenberg, 2010, p. 8).

La comparaison que Goldenberg fait entre la syntaxe minimaliste des wikis et un début de programmation est ici intéressante. Elle rejoint notre questionnement sur les frontières de la notion de code source. Comme nous l'avons noté dans le cours de nos chapitres, certains aspects du code source des projets – en particulier les squelettes dans SPIP – ont un statut similaire à celui décrit par Goldenberg : « un début de programmation ». D'une certaine manière, nous pourrions appeler « code source » le texte d'un wiki avec lequel l'utilisateur interagit pour élaborer une page de wiki. Comme le montre la figure suivante (7.1), ce texte renferme également des « balises » qui constituent en quelque sorte un début de programmation.

```
{{Sommaire limité au niveau 2}}

== Étymologie de Québec ==
{{Article détaillé|Québec (ville)#Histoire du nom « Québec »{{!}}Histoire du nom « Québec »}}

Le vocable « Québec » — signifiant « là où le fleuve se rétrécit » en langue [[algonquin]]e — était employé par les [[Algonquins (Amérindiens)|Algonquins]], les [[Cris]] et les [[Micmacs]]<ref name="Hamilton225">{{ouvrage |langue=en |prénom1=William B. |nom1=Hamilton |titre=The Macmillan book of Canadian place names |éditeur=Macmillan of Canada |lieu=Toronto |année=1978 |passage=225}}, cité dans {{lien weblurl=http://geonames.nrcan.gc.ca/education/prov_f.php#qc |titre=Noms géographiques canadiens |site=ministère des Ressources naturelles |auteur=Canada |consulté le=21 janvier 2009}}.</ref> pour désigner le rétrécissement du
```

Figure 7.1 : L'écriture d'un wiki
Extrait de la page *Québec* de Wikipédia : <http://fr.wikipedia.org/w/index.php?title=Québec&action=edit> (consulté le 6 décembre 2011).

Mentionnons que plusieurs initiatives contemporaines rendent davantage poreuse la frontière entre le code source et le wiki. C'est le cas par exemple de ikiwiki[200], un logiciel qui se décrit comme un *compilateur de wiki* (« a wiki compiler »), qui convertit des pages wikis en pages html propres à être publiées sur un site web. Contrairement aux wikis conventionnels, qui enregistrent leur page dans une base de données, ikiwiki s'appuie plutôt sur des systèmes de gestion de versions tels que Subversion et Git pour gérer l'historique des versions d'une page. Autant la notion de *compilateur*, mise de l'avant pour décrire la fonctionnalité du logiciel, que les dispositifs de gestion des versions avec lesquels l'écriture des pages de wikis s'articule, montrent bien la proximité des pages de wikis avec l'artefact code source, tel que nous l'avons étudié dans cette thèse[201].

200 <http://ikiwiki.info> (consulté le 24 novembre 2011). Merci à Antoine Beaupré pour cette référence.

201 Un billet de blogue décrit d'ailleurs les avantages de ikiwiki en parlant des « sources du wiki » : « Les sources du wiki peuvent être dans un dépôt tel sur Subversion, GIT, Mercurial, etc., ce qui permet un suivi efficace des changements ».
<http://olivier.dossmann.net/joueb/archives/2008/10/09/ikiwiki_un_moteur_wiki_pas_comme_les_autres/> (consulté le 24 novembre 2011).

Code source et rédaction de la thèse

Si la forme wiki est une base intéressante de comparaison avec le code source, cette comparaison pourrait à notre avis être étendue à tout type de document numérique. À plusieurs moments dans la rédaction de notre thèse sont apparues des similitudes entre le document sur lequel nous avons travaillé et les formes que prenait le code source dans les projets étudiés. D'une part, comme nous l'avons fait ressortir maintes fois dans cette thèse, il est difficile d'établir de façon très précise les frontières du code source. Dans la perspective de l'usage d'un traitement de texte, par exemple pour rédiger cette thèse de doctorat, le code source pourrait en quelque sorte constituer notre document de travail, en format .odt (ou .doc). Ceci correspondrait à la définition que donne la licence publique générale du code source comme la forme préférée d'un travail pour faire des modifications sur celui-ci (« the preferred form of the work for making modifications to it[202] »). Le code « exécutable » serait quant à lui le document PDF produit ou le document officiel envoyé à la bibliothèque. Dans cette perspective, déposer cette thèse sous une licence GPL impliquerait de rendre disponibles, non seulement le PDF, mais également le document en format .odt pour faciliter un travail subséquent.

L'écriture d'une thèse de doctorat – à l'aide d'un document numérique – renvoie également à des aspects de « programmabilité » ou de « configurabilité » qui sont similaires à ceux de la programmation. Par exemple, les renvois à l'intérieur d'un texte vers une figure ou une référence bibliographique sont des aspects qui relèvent davantage de la « programmabilité » ou de la « configurabilité » de l'artefact que de la simple écriture. La mise en forme, à l'aide du gestionnaire de styles, relève également d'une certaine « configurabilité » du document. Encore une fois, ces aspects de configurabilité sont perdus dans la diffusion du seul document PDF, sans parler de la diffusion de la thèse en format papier. Il en est de même pour les commentaires et les outils de révision[203].

---

202 <http://www.gnu.org/licenses/gpl.html> (consulté le 6 décembre 2011).

203 Un aspect que nous n'avons pas abordé dans cette thèse concerne la programmation dite « littéraire ». En effet, si les traitements de texte « ce que vous voyez est ce que vous aurez » sont aujourd'hui très populaires, de nombreuses personnes ont encore recours à des outils tels que *Latex* pour élaborer leurs documents. Ces outils sont en effet des langages de programmation dits « littéraires » dont le rôle est surtout de faciliter la mise en forme des documents.

Finalement, mentionnons ici la manière dont cette étude a inspiré notre propre pratique. Ainsi, à un certain point dans notre rédaction, nous avons décidé d'utiliser le logiciel Subversion (SVN) et la transaction du commit, afin de gérer les différentes versions de nos chapitres. De plus, nous avons mis en place un système de suivi de tâches, à la manière du système de tickets dans les projets étudiés, de façon à faire ressortir les différentes tâches qui restaient à faire dans la thèse. Si les pratiques d'écriture distribuée, contributive et collaborative sont utiles dans le cadre de la rédaction individuelle, elles le sont certainement dans le cadre d'une pratique collective.

7.3 En guise de conclusion. Le code source, ou le lien social dans la machine

À la fin du premier chapitre, nous avons cité Serge Proulx (2012), qui mentionnait la pertinence des études sur le code informatique, par la manière dont elles abordent le noyau des infrastructures technologiques avec lesquelles nous interagissons quotidiennement. Selon Proulx, l'étude du code informatique (et par conséquent, du code source) permet en quelque sorte d'ouvrir la boîte noire des technologies logicielles et d'appréhender les configurations complexes et changeantes qui se situent dans le « noyau » des technologies logicielles et des infrastructures technologiques. À l'instar de ces propos de Proulx, nous avons tenté, dans le cadre de cette thèse, d'ouvrir la boîte noire des infrastructures technologiques, en analysant la manière dont le code source est fabriqué, mais surtout, en portant une attention particulière aux formes et statuts du code source dans les projets étudiés, aux autorisations avec lesquelles le code source est articulé, de même que la négociation de ces formes, statuts et autorisations.

Notre intention dans cette thèse va cependant plus loin que l'objectif d'analyser le noyau des technologies avec lesquels nous interagissons. Notre étude partage plutôt le même esprit que celui de Simondon de « susciter une prise de conscience du sens des objets techniques », et mettre de l'avant la manière dont le code source est en lui-même, non seulement le « noyau » des infrastructures technologiques, mais également pour certains acteurs, le lieu même de l'interaction et de l'engagement. L'un des acteurs explique ainsi son engagement avec le code source dans un extrait que nous avons identifié en toute fin de thèse :

> On a une ingénierie informatique, qui dit plus ou moins son nom. On a une organisation des serveurs sur lesquels se produit effectivement la construction de ces codes. Et s'échange, soit dans le code, soit dans les commentaires, soit sur la documentation, soit dans le bouillis autour de ça. On a un ensemble d'une quinzaine de sites, tenus par les uns, par certaines parties, et certaines autres parties. C'est structuré par des codes d'accès, des autorisations. Qu'est-ce qui fait que quelqu'un devient un moment éligible à tel ou tel cercle ? À tel ou tel petit groupe ? Parce qu'il a manifesté son intérêt et son talent. Car les autres ont décidé de coopter ce type [...]. Or, tout ça est organisé autour du code. Autour du regard que l'on pose sur ce code.
>
> [...]
>
> Comment je me suis mis à parler à toto [nom fictif] ? Parce que sa fonction, elle merdait... Sa fonction merde, je fais une correction de sa fonction, et je vois que celui qui l'a développée s'appelle [nom fictif]. Je lui envoie un mail en disant j'ai fait ça, j'ai fait ça, etc. Et il me répond : « c'est vachement bien », etc. Donc, la méthode par laquelle je rencontre des gens, la méthode par laquelle se structurent les relations humaines que nous avons là-dedans, c'est le code qui nous amène à... c'est le code qui nous amène à rencontrer les gens (sf07).

Ainsi, le code source ne doit pas être appréhendé simplement comme ce qui se trouve « à l'intérieur » de la boîte noire que sont les technologies dont nous faisons usage. Nous avons au contraire voulu faire contre-pied à cette perspective, présente dans bon nombre d'études sur les logiciels libres et sur l'usage des technologies de l'information, de ne s'intéresser au code source qu'en tant qu'il ne relève du domaine de la conception, du domaine de la technique, de ce qui, pour reprendre les termes de Simondon, n'a pas de signification. Nous avons au contraire voulu montrer dans cette thèse la complexité de cet artefact aux frontières ambiguës, qui consiste en un réseau complexe de fichiers, de textes, d'images et d'autres médias et qui réussit à mobiliser, pour exister, l'engagement de centaines d'acteurs humains.

Politiquement, notre thèse s'inscrit également dans l'esprit du mouvement du logiciel libre, en s'opposant à l'effacement du code source de la réalité humaine. Nous avons en effet expliqué, dans le premier chapitre, la manière dont les militants du logiciel libre considèrent que la restriction légale de l'accès au code source constitue une atteinte à la liberté d'expression. Il s'agit donc, comme nous l'avons également exposé au premier chapitre, de mettre de l'avant la dimension expressive du code source, c'est-à-dire de montrer que l'étude du code source n'est pas seulement pertinente à analyser parce qu'elle concerne le noyau des infrastructures technologiques, mais également, et peut-être surtout, parce que le code source est appréhendé

par les acteurs comme une forme d'expression qui doit être respectée en elle-même. À cet effet, mentionnons encore une fois ces sites web, tels que GITHub ou encore ohloh.net, qui facilitent le partage du code source dans une dynamique qui rappelle les nombreux sites de type web 2.0, comme Flicker ou Twitter. Dans le cas de ces sites web (tels que GITHub ou ohloh.net), le code source prend toute son importance, non pas en tant qu'infrastructure de la collaboration, mais plutôt en tant que contenu de la collaboration, qui est échangé et discuté entre un grand nombre d'acteurs. C'est dans cette perspective que l'on peut le mieux saisir ce caractère expressif du code source, et la manière dont il peut être appréhendé comme un média, au même titre que d'autres formes médiatiques du web 2.0 tels que les blogues, pages de wikis, tweets, images, etc.

Comme le souligne cette citation de Suchman que nous avons placée en exergue de ce chapitre, plutôt que d'appréhender les interactions humain-machine sous la forme d'une conversation qui se déroulerait par l'intermédiaire d'une interface changeante et transparente, il semble davantage pertinent de saisir les capacités dynamiques que les nouveaux médias permettent, de reconfigurer constamment et rapidement les relations entre humains et machines. C'est dans cette perspective que le code source nous apparaît un objet pertinent à analyser en communication, parce qu'il semble constituer de plus en plus une forme médiatique originale et spécifique de l'ère numérique, à travers laquelle les acteurs humains communiquent et coopèrent entre eux.

RÉFÉRENCES

Abelson, Harold et Gerald Jay Sussman. 1996. *Structure and Interpretation of Computer Programs*. 2ième éd. Cambridge (Mass.) : The MIT Press.

Aigrain, Philippe. 2005. *Cause commune : l'information entre bien commun et propriété*. Paris : Fayard. En ligne <http://www.causecommune.org/download/>. Consulté le 9 novembre 2007.

Akrich, Madeleine. 1993. « Les formes de la médiation technique ». *Réseaux*, no 60, p. 87-98.

Akrich, Madeleine, Michel Callon et Bruno Latour. 2006. *Sociologie de la traduction. Textes fondateurs*. Paris : École des Mines de Paris.

Akrich, Madeleine et Bruno Latour. 1992. « A Summary of a Convenient Vocabulary for the Semiotics of Human and Nonhuman Assemblies ». In *Shaping Technology/Building Society. Studies in Sociotechnical Change*, sous la dir. de Wiebe E. Bijker et John Law, p. 259-264. Cambridge (Mass.) : The MIT Press.

Alexander, Christopher. 1977. *A Pattern Language : Towns, Buildings, Construction*. New York : Oxford University Press.

Appadurai, Arjun (dir. publ.). 1988. *The Social Life of Things : Commodities in Cultural Perspective*. Cambridge (Angleterre) : Cambridge University Press.

Arendt, Hannah. 1972. *La crise de la culture huit : exercices de pensée politique*. Paris : Gallimard.

Arns, Inke. 2005. « Code as Performative Speech Act ». *Artnodes* (juillet). En ligne <http://www.uoc.edu/artnodes/eng/art/arns0505.pdf>. Consulté le 7 décembre 2009.

Association française de normalisation. 1986. *Vocabulaire international de l'informatique*. Paris-La Défense : AFNOR.

Astier, Éric. 2001. *Dictionnaire des technologies de l'information*. Paris : Foucher.

Auray, Nicolas. 2000. « Politique de l'informatique et de l'information. Les pionniers de la nouvelle frontière électronique ». Thèse de doctorat, Paris, EHESS. En ligne <http://ses.telecom-paristech.fr/auray/Auray%20These.pdf>. Consulté le 30 mai 2011.

———. 2007. « Le modèle souverainiste des communautés en ligne : Impératif participatif et désacralisation du vote. ». *Hermès*, no 47, p. 137-144.

———. 2009. « Le travail en réseau : de Wikipédia à Linux ». In *L'évolution des cultures numériques, de la mutation du lien social à l'organisation du travail*, sous la dir. de Christian Licoppe, p. 124-134. Limoges (France) : Fyp Editions.

Austin, John Langshaw. 1975. *How to Do Things With Words*. 2e éd. Cambridge (Mass.) : Harvard University Press.

———. 1991. *Quand dire, c'est faire*. Paris : Points (Seuil).

Backus, J. W., R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, et al. 1957. « The FORTRAN automatic coding system ». In *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability* (Los Angeles (California)), p. 188-198. En ligne <http://portal.acm.org/citation.cfm?id=1455567.1455599>. Consulté le 22 juillet 2010.

Barad, Karen. 2003. « Posthumanist Perfomativity : Toward an Understanding of How Matter Comes to Matter ». *Signs : Journal of Women in Culture and Society*, vol. 28, no 3, p. 801-831.

Barcellini, F., F. Détienne, J. M Burkhardt et W. Sack. 2006. « Visualizing Roles and Design Interactions in an Open Source Software Community ». In *Supporting the Social Side of Large Scale Software Development* (Banff, Californie, Banff, 4 novembre 2006).

Barcellini, Flore. 2008. « Conception de l'artefact, conception du collectif. Dynamique d'un processus de conception ouvert et continu dans une communauté de développement de logiciels libres ». Thèse de doctorat, Paris, Conservatoire national des arts et métiers. En ligne <http://ergonomie.cnam.fr/equipe/barcellini/these.pdf>. Consulté le 15 novembre 2012.

Barcellini, Flore, Françoise Détienne et Jean-Marie Burkhardt. 2007. « Conception de logiciels libres : enjeux pour l'ergonomie et rôle des utilisateurs dans le processus de conception ». In *XXXXIIème congrès de la Société Ergonomique de Langue Française* (Saint-Malo, France, 5-7 septembre 2007), p. 43-52. En ligne <http://hal.inria.fr/index.php?halsid=am61ru71asnprirn3dkvg42ps4&view_this_doc=inria-00177840&version=1>. Consulté le 8 juillet 2009.

Beck, Kent et Ward Cunningham. 1987. « Using Pattern Languages for Object-Oriented Programs ». In *OOPSLA-87 Workshop on the Specification and Design for Object-Oriented Programming* (Orlando, Floride, 17 septembre 1987).

Becker, Howard S. 1986. *Writing for Social Scientists: How to Start and Finish Your Thesis, Book, or Article*. Chicago : University of Chicago Press.

Bijker, Wiebe E., Thomas Parke Hughes et T. J. Pinch (dir. publ.). 1987. *The Social Construction of Technological Systems : New Directions in the Sociology and History of Technology*. Cambridge (Mass.) : MIT Press.

Bitter, Gary G. 1992. *Macmillan Encyclopedia of Computers*. New York : Macmillan.

Blondeau, Olivier. 2004. « Celui par qui le code est parlé ». *boson2x*. En ligne <http://www.boson2x.org/spip.php?article94>. Consulté le 21 décembre 2011.

———. 2005. « Des hackers aux cyborgs : le bug simondonien ». *Revue Multitude*, no 18. En ligne <http://multitudes.samizdat.net/Des-hackers-aux-cyborgs-le-bug.html>.

———. 2007. « Les multitudes seront « syndiquées » ou ne seront pas : la politique d'agrégation ». In *Devenir média, l'activisme sur Internet entre défection et expérimentation*, p. 309-365. Paris : Editions Amsterdam. En ligne <http://www.devenirmedia.net/DevenirMedia.pdf>.

Bloor, David. 1991. *Knowledge and Social Imagery*. 2ième édition. Chicago : University of Chicago Press.

Botto, Francis. 1999. *Dictionary of Multimedia and Internet Applications: A Guide for Developers and Users*. 1re éd. Chichester (UK) : Wiley.

Bourdieu. 2000. *Propos sur le champ politique*. Lyon : Presses Universitaires de Lyon.

Bourdieu, Pierre. 1975. « Le langage autorisé. Note sur les conditions sociales de l'efficacité du discours rituel ». *Actes de la recherche en sciences sociales*, vol. 1, no 5-6, p. 183-190.

Bourdieu et Loïc J. D Wacquant. 1992. *Réponses pour une anthropologie réflexive*. Paris : Éditions du Seuil.

Bowker, G.C., K. Baker, F. Millerand et D. Ribes. 2010. « Toward information infrastructure studies: Ways of knowing in a networked environment ». In *International Handbook of Internet Research*, sous la dir. de Jeremy Hunsinger, Lisbeth Klastrup et Matthew Allen, p. 97–117.

Bowker, Geoffrey. 1994. « Information Mythology and Infrastructure ». In *Information Acumen: The Understanding and Use of Knowledge in Modern Business*, sous la dir. de Lisa Bud-Frierman, p. 231-347. New York : Routledge.

Bowker, Geoffrey C. et Susan Leigh Star. 2000a. *Sorting Things Out : Classification and Its Consequences*. Cambridge (Mass.) : The MIT Press.

Bowker, Geoffrey et Susan Leigh Star. 2000b. « Invisible Mediators of Action : Classification and the Ubiquity of Standards ». *Mind, Culture, and Activity*, vol. 7, no 1, p. 147-163.

Breton, Philippe et Serge Proulx. 2002. *L'explosion de la communication : à l'aube de XXIe siècle*. Nouv. éd. Montréal : Boréal.

Brousseau, Eric et Frédéric Moatty. 2003. « Perspectives de recherche sur les TIC en sciences sociales : Les passerelles interdisciplinaires d'Avignon ». *Sciences de la société*, no 59, p. 3-33.

Buswell, Evan. 2009. « Truth and Command in the Language of Code: (code := meaning) == (code := action)? ». In *What is Code? What is Coding? Emerging STS Approaches in Studying Computer Code. Society for Social Studies of Science 2009 Conference* (Washington DC, 28 octobre au 1er novembre 2009).

Butler, Judith. 1990. *Gender Trouble : Feminism and the Subversion of Identity*. New York : Routledge.

———. 1997. *Excitable Speech : A Politics of the Performative*. 1re éd. New York : Routledge.

———. 2004. *Le pouvoir des mots : Politique du performatif*. Paris : Editions Amsterdam.

Byfield, Bruce. 2006. « FSF reaches out to social activists ». *NewsForge*, 20mai. En ligne <http://software.newsforge.com/article.pl?sid=06/08/31/158231>. Consulté le 7 janvier 2007.

Callon, Michel. 2006a. *What does it mean to say that economics is performative?* Papiers de recherche du CSI. Paris : Centre de Sociologie de l'Innovation (CSI), Mines ParisTech. En ligne <http://ideas.repec.org/p/emn/wpaper/005.html>. Consulté le 14 février 2009.

———. 2006b. « Sociologie de l'acteur réseau ». In *Sociologie de la traduction. Textes fondateurs*, sous la dir. de Madeleine Akrich, Michel Callon et Bruno Latour, p. 267-276. Paris : Presses de l'École des Mines de Paris.

———. 2007. « What does it mean to say that economics is performative ». In *Do economists make markets*, sous la dir. de Mackenzie, Muniesa et Lucia Siu, p. 311–357. Princeton : Princeton University Press.

Callon, Michel et Bruno Latour. 1986. « Les paradoxes de la modernité : comment concevoir les innovations ? ». *Prospective et santé*, no 36, p. 13-25.

———. 1991. « Introduction ». In *La Science telle qu'elle se fait*, p. 7-36. Paris : La Découverte.

Cardon, Dominique. 2005. « De l'innovation ascendante. Entrevue réalisée par Hubert Guillaud ». *InternetActu*. En ligne <http://www.internetactu.net/?p=5995>. Consulté le 9 novembre 2007.

———. 2008. « Le design de la visibilité : un essai de typologie du web 2.0 ». *InternetActu*. En ligne <http://www.internetactu.net/2008/02/01/le-design-de-la-visibilite-un-essai-de-typologie-du-web-20/>. Consulté le 31 août 2009.

Certeau, Michel de. 1990. *L'invention du quotidien*. Paris : Galimard.

Cohn, Marisa Leavitt. 2009. « Code as Conversation in Agile software development ». In *What is Code? What is Coding? Emerging STS Approaches in Studying Computer Code. Society for Social Studies of Science 2009 Conference* (Washington DC, 28 octobre au 1er novembre 2009).

Coleman, Gabriella. 2003. « The (Copylefted) Source Code for the Ethical Production of Information Freedom ». *Sarai Reader 2003 : Shaping Technologies*, p. 297-302.

———. 2009. « CODE IS SPEECH : Legal Tinkering, Expertise, and Protest among Free and Open Source Software Developers ». *Cultural Anthropology*, vol. 24, no 3, p. 420-454.

Collins, Harry et Steven Yearley. 1992. « Epistemological Chicken ». In *Science as Practice and Culture*, sous la dir. de Andrew Pickering, p. 301-326. Chicago : University of Chicago Press.

Coris, Marie. 2006. « Chronique d'une absorption par la sphère marchande : les Sociétés de Services en Logiciels Libres. ». *Gérer & Comprendre*, no 84, p. 12-24.

Couture, Stephane. 2006. « La construction du modèle du libre ». In *Colloque « logiciel libre en tant que modèle d'innovation sociotechnique », 74e congrès de l'ACFAS* (Université McGill, Montréal, 16 mai 2006). En ligne <http://cmo.uqam.ca/node/62>. Consulté le 8 décembre 2008.

Couture, Stéphane. 2007. « Logiciel libre, activité technique et engagement politique: la construction du projet GNU en Argentine ». Mémoire de maîtrise, Montréal, Université du Québec à Montréal.

———. 2008. « Gilberto Gil: politiser la culture numérique - ». *Journal des Alternatives*, 28février. En ligne <http://journal.alternatives.ca/fra/journal-alternatives/publications/archives/2008/volume-14-no-06-mars/article/gilberto-gil-politiser-la-culture>. Consulté le 24 avril 2011.

Couture, Stéphane et Marisa Leavitt Cohn. 2009. « What is Code? What is Coding? Emerging STS approaches in studying computer code ». In *Society for Social Studies of Science 2009 Conference* (Washington DC, 28 octobre au 1er novembre 2009).

Couture, Stéphane, Christina Haralanova, Sylvie Jochems et Serge Proulx. 2010. *Un portrait de l'engagement pour les logiciels libres au Québec*. Montréal : Centre de recherche sur la science et la technologie (CIRST).

Couture, Stéphane et Serge Proulx. 2008. « Les militants du code ». In *L'action communautaire à l'ère du numérique*, p. 14-35. Montréal : Presses de l'Université du Québec.

Debaise, Dider. 2005. « Le langage de l'individuation ». *Revue Multitudes*, no 18. En ligne <http://multitudes.samizdat.net/Le-langage-de-l-individuation>. Consulté le 31 janvier 2011.

Deissenboeck, Florian et Markus Pizka. 2006. « Concise and Consistent Naming ». *Software Quality Journal*, vol. 14, no 3, p. 261-282.

Demazière, Didier, François Horn et Marc Zune. 2006. « Dynamique de développement des communautés du logiciel. Conditions d'émergence et régulations des tensions ». *Revue Terminal*, no 97-98, p. 71-84.

———. 2007a. « Des relations de travail sans règles ? ». *Sociétés contemporaines*, vol. 66, no 2, p. 101.

———. 2007b. « The Functioning of a Free Software Community : Entanglement of Three Regulation Modes - Control, Autonomous and Distributed ». *Science Studies*, vol. 20, no 2, p. 34-54.

———. 2008. « Les mondes de la gratuité à l'ère du numérique : une convergence problématique sur les logiciels libres ». *Revue Française de Socio-économie*, no 1, p. 47-65.

Denis, Jérôme. 2006. « Les nouveaux visages de la performativité ». *Études de communication*, no 29, p. 7-24.

———. 2007. « La prescription ordinaire. Circulation et énonciation des règles au travail ». *Sociologie du Travail*, vol. 49, no 4, p. 496-513.

Denis, Jérôme et David Pontille. 2010a. « Performativité de l'écrit et travail de maintenance ». *Réseaux*, vol. 163, no 5, p. 105-130.

———. 2010b. *Petite sociologie de la signalétique : Les coulisses des panneaux du métro*. Paris : Presses de l'Ecole des mines.

Deslauriers, J.-P. et M. Kérisit. 1997. « Le devis de recherche qualitative ». In *La recherche qualitative. Enjeux épistémologiques et méthodologiques*, sous la dir. de J. Poupart, J.-P. Deslauriers, L.-H. Groulx, A. Laperrière, R. Mayer et A.P. Pires, p. 85-111. Montréal : Gaëtan Morin Éditeur.

Détienne, Françoise. 1998. *Génie logiciel et psychologie de la programmation*. Paris : Hermès.

Dewey, John. 1896. « The Reflex Arc Concept in Psychology ». *Psychological Review*, no 3, p. 357-370.

Doueihi, Milad. 2008. *La Grande Conversion numérique*. Paris : Seuil.

Downing, Douglas, Michael A. Covington et Melody Mauldin Covington. 2000. *Dictionary of Computer and Internet Terms*. 7e éd. New York : Barron's Educational Series, Inc.

Doyle, Richard. 1997. *On Beyond Living: Rhetorical Transformations of the Life Sciences*. 1re éd. Stanford (Calif.) : Stanford University Press.

Feenberg, Andrew. 2004. *(Re)penser la technique : vers une technologie démocratique*. Paris : La Découverte/M.A.U.S.S.

Flichy, Patrice. 1995. « L'anthropologie de la technique. Penser ensemble le technique et le social ». In *L'innovation technique*, p. 75-108. Paris : La Découverte.

Fowler, Martin. 2003. *Patterns of Enterprise Application Architecture*. Boston (Mass.) : Addison-Wesley.

Fraenkel, Béatrice. 2006. « Actes écrits, actes oraux : la performativité à l'épreuve de l'écriture ». *Études de communication*, no 29, p. 69-93.

———. 2007. « Actes d'écriture : quand écrire c'est faire ». *Langage et société*, vol. 121-122, no 3, p. 101.

———. 2008. « La signature : du signe à l'acte ». *Sociétés & Représentations*, vol. 25, no 1, p. 13.

Fujimura, Joan H. 1992. « Crafting Science: Standardized Packages, Boundary Objects, and « Translation » ». In *Science as Practice and Culture*, sous la dir. de Andrew Pickering, p. 168-211. Chicago : University of Chicago Press.

Fuller, Matthew. 2008. *Software Studies: A Lexicon*. Cambridge (Mass.) : The MIT Press.

Gamma, Erich, Richard Helm, Ralph Johnson et John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1re éd. Reading (Mass.) : Addison-Wesley Professional.

Geertz, Clifford. 1998. « La description dense. Vers une théorie interprétative de la culture ». *Enquête. Cahiers du Cercom*, vol. 1, no 6, p. 73-105.

Geiger, R. Stuart et David Ribes. 2010. « The Work of Sustaining Order in Wikipedia ». In *Proceedings of the 2010 ACM conference on Computer supported cooperative work - CSCW '10* (Savannah, Georgia, 6–10 février 2010,), p. 117-126. En ligne <http://portal.acm.org/citation.cfm?id=1718941>. Consulté le 30 novembre 2010.

Gell, Alfred. 1998. *Art and Agency : An Anthropological Theory*. Oxford ; New York : Clarendon Press.

Ghosh, Rishab Aiyer, Gregorio Robles et Ruediger Glott. 2002. *Floss Final Report. Software Source Code Survey*. University of Maastricht, The Netherlands : International Institute of Infonomics. En ligne <http://www.flossproject.org/report/FLOSS_Final5all.pdf>. Consulté le 8 décembre 2008.

Gibson, William. 2008a. *Code Source*. Vauvert (France) : Au Diable Vauvert.

———. 2008b. *Spook Country*. Reprint. Berkley (TRD).

Gieryn, Thomas F. 1983. « Boundary-Work and the Demarcation of Science from Non-Science : Strains and Interests in Professional Ideologies of Scientists ». *American Sociological Review*, no 48, p. 781-795.

Ginguay, Michel. 1987. *Dictionnaire d'informatique bureautique, telematique, micro-informatique : anglais-francais*. 9e ed. rev. et augm. Paris : Masson.

———. 2005. *Dictionnaire Anglais/Français Informatique*. 14e éd. Paris : Dunod.

Godbout, Jacques T. 2000. *Le don, la dette et l'indentité : homo donator versus homo oeconomicus*. Montréal : Éditions du Boréal.

Goldenberg, Anne. 2010. « La négociation des contributions dans les wikis publics : légitimation et politisation de la cognition collective ». Thèse de doctorat, Montréal, Université du Québec à Montréal.

Graham, Paul. 2004. *Hackers and Painters: Big Ideas from the Computer Age*. Sebastopol (Calif.) : O'Reilly Media, Inc.

Gramaccia, G. 2001. *Les actes de langage dans les organisations*. Paris : L'Harmattan.

Hall, Stuart. 1994. « Codage/décodage ». *Réseaux*, no 68, p. 27-39.

Hammersley, Martyn et Roger Gomm. 2000. « Introduction ». In *Case Study Method. Key Isssues, Key Texts*, sous la dir. de Roger Gomm, Martyn Hammersley et Peter Foster, p. 1-16. London : Sage Publications.

Haralanova, Kristina. 2010. « L'apport des femmes dans le développement du logiciel libre ». Mémoire de maîtrise, Montréal, Université du Québec à Montréal.

Haraway, Donna. 1985. « A Cyborg Manifesto : Science, Technology, and Socialist-Feminism in the Late 20th Century ». *Socialist Review*, no 80, p. 65-108.

———. 1988. « Situated Knowledge : The Science Question in Feminism as a Site of Discourse on the Privilege of Partial Perspective ». *Feminist Studies*, vol. 14, no 3, p. 575-599.

———. 1991. *Simians, Cyborgs, and Women : The Reinvention of Nature*. New York (NY) : Routledge.

———. 1997. *Modest-Witness@Second-Millennium.FemaleMan-Meets-OncoMouse : feminism and technoscience*. New York : Routledge.

Haring, Kristen. 2009. « Handmade Binary : The Codes of Telegraphers and of Knitters, Fit Together ». In *What is Code? What is Coding? Emerging STS Approaches in Studying Computer Code. Society for Social Studies of Science 2009 Conference* (Washington DC, 28 octobre au 1er novembre 2009).

Hennion, Antoine. 2005. « Pour une pragmatique du goût ». *Working papers du Centre de sociologie de l'innovation*, no 1. En ligne <http://www.csi.ensmp.fr/Items/WorkingPapers/Download/DLWP.php?wp=WP_CSI_001.pdf>.

———. 2007. *La passion musicale : Une sociologie de la médiation*. 2ième édition revue et corrigée. Paris : Métailié.

Herrenschmidt, Clarisse. 2007. *Les trois écritures. Langue, nombre, code*. Paris : Galimard.

Hess, David J. 2001. « Ethnography and the Development of Science and Technology Studies ». In *Sage Handbook of Ethnography*, sous la dir. de Paul Atkinson, Amanda Coffey, Sara Delamont, John Lofland et Lyn Loftland, p. 234-245. Thousand Oaks (Calif.) : SAGE Publications.

Hine, Christine. 2000. *Virtual Ethnography*. Thousand Oaks (Calif.) : Sage Publications Ltd.

———. 2005. *Virtual Methods : Issues in Social Research on the Internet*. London : Berg Publishers.

———. 2007. « Multi-sited Ethnography as a Middle Range Methodology for Contemporary STS ». *Science Technology Human Values*, vol. 32, no 6, p. 652-671.

Hippel, Eric von. 2005. *Democratizing Innovation*. Cambridge (Mass.) : The MIT Press. En ligne <http://web.mit.edu/evhippel/www/democ1.htm>. Consulté le 30 avril 2012.

Illingworth, Valérie (dir. publ.). 1991. *Dictionnaire d'informatique*. Paris : Hermann.

Jensen, C.B. 2004. *Researching Partially Existing Objects: What is an Electronic Patient Record? Where do you find it? How do you study it?* Aarhus : Centre for STS Studies. En ligne <http://imv.au.dk/sts/arbejdspapirer/WP4.pdf>.

Jensen, J. 1965. « Generation of Machine Code in ALGOL Compilers ». *BIT Numerical Mathematics*, vol. 5, no 4, p. 235–245.

Joerges, B. 1999. « Do Politics Have Artefacts? ». *Social Studies of Science*, vol. 29, no 3, p. 411-431.

Jones, Duncan (réal.). 2011. *Source Code*. Film. Los Angeles (Calif.) : The Mark Gordon Company, Vendôme Pictures, 93 min.

Kane, Carolyn. 2009. « 'Programming the Beautiful' : Digital Codes Color and Aesthetic Transformations in Early Computer Art ». In *What is Code? What is Coding? Emerging STS Approaches in Studying Computer Code. Society for Social Studies of Science 2009 Conference* (Washington DC, 28 octobre au 1er novembre 2009).

Kelty, Christopher. 2008. *Two Bits: The Cultural Significance of Free Software*. Durham (NC) : Duke University Press.

Kittler, Friedrich. 2008. « Code (or, How You Can Write Something Differently) ». In *Software Studies: A Lexicon*, sous la dir. de Matthew Fuller, p. 236-243. Cambridge (MA) : The MIT Press.

Knouf, Nick. 2009. « Questioning « Openness » Through a Study of Discussions Surrounding the OLPC XO Laptop ». In *What is Code? What is Coding? Emerging STS Approaches in Studying Computer Code. Society for Social Studies of Science 2009 Conference* (Washington DC, 28 octobre au 1er novembre 2009).

Knuth, Donald Ervin. 1968. *The Art of Computer Programming*. Reading (Mass.) : Addison-Wesley Pub. Co.

———. 1974. « Computer Programming as an Art ». *Communications of the ACM*, vol. 17, no 12, p. 667 - 673.

Krippendorff, Klaus. 1993. « Major Metaphors of Communication and some Constructivist Reflections on their Use ». *Cybernetics & Human Knowing*, vol. 2, no 1, p. 3-25.

Kristoffersen, Steinar. 2006. « Designing a Program. Programming the Design ». *TeamEthno-online Issue*, vol. 2, p. 34-51.

Krysia, Josia et Sedek Grzesiek. 2008. « Source Code ». In *Software Studies : A Lexicon*, sous la dir. de Matthew Fuller, p. 236-243. Cambridge (Mass.) : The MIT Press.

Latour, Bruno. 1985. « Les « Vues » de l'esprit. Une introduction à l'anthropologie des sciences et des techniques ». *Culture Technique*, no 14, p. 4-29.

———. 1991. *Nous n'avons jamais été modernes. Essai d'anthropologie symétrique*. Paris : La Découverte.

———. 1992a. *Aramis ou L'amour des techniques*. Paris : La Découverte.

———. 1992b. « Where Are the Missing Masses? The Sociology of a Few Mundane Artifacts ». In *Shaping Technology/Building Society. Studies in Sociotechnical Change*, sous la dir. de Wiebe E Bijker et John Law, p. 225-258. Cambridge (Mass.) : The MIT Press.

———. 1993. *We Have Never Been Modern*. Cambridge (Mass.) : Harvard University Press.

———. 2000. « Morale et technique : la fin des moyens ». *Réseaux*, vol. 18, no 100, p. 29-58.

———. 2001. « Le dédale de la médiation technique ». In *L'espoir de Pandore*, p. 215-260. Paris : La Découverte.

———. 2004. *La fabrique du droit : Une ethnographie du Conseil d'Etat*. Paris : La Découverte.

———. 2006. *Changer de société - Refaire de la sociologie*. Paris : La Découverte.

———. 2010. « Prendre le pli des techniques ». *Réseaux*, vol. 163, no 5, p. 11-31.

Latour, Bruno et Steve Woolgar. 1979. *Laboratory Life the Social Construction of Scientific Facts*. Beverly Hills (Calif.) : Sage.

———. 1988. *La vie de laboratoire : la production des faits scientifiques*. Paris : La Découverte.

Latzko-Toth, Guillaume. 2010. « La co-construction d'un dispositif sociotechnique de communication : le cas de l'internet relay chat ». Thèse de doctorat, Montréal, Université du Québec à Montréal.

Law, John (dir. publ.). 1991. *A Sociology of Monsters: Essays on Power, Technology, and Domination*. London : Routledge.

Law, John et John Hassard (dir. publ.). 1999. *Actor network theory and after*. Oxford : Blackwell.

Lejeune, Christophe. 2008. « Au fil de l'interprétation. L'apport des registres aux logiciels d'analyse qualitative ». *Swiss Journal Of Sociology*, vol. 34, no 3, p. 593-603.

Lessig, Lawrence. 1998. « Open Code and Open Societies : Values of Internet Governance ». *Chicago-Kent Law Review*, vol. 74, p. 101-116.

———. 2000. *Code and Other Laws of Cyberspace*. New York (NY) : Basic Books.

———. 2006. *Code : Version 2.0*. New York (NY) : Basic Books.

Libre-Fan. 2008. « SPIP, dépassé? ». *Libre-Fan*. En ligne <http://librefan.eu.org/node/138>. Consulté le 25 février 2011.

Licoppe, Christian. 2008. « Dans le « carré de l'activité » : perspectives internationales sur le travail et l'activité ». *Sociologie du Travail*, vol. 50, no 3, p. 287-302.

Licoppe, Christian, Serge Proulx et Renato Cudicio. 2010. « L'émergence d'un nouveau genre communicationnel dans les organisations fortement connectées : les « questions rapides » par messagerie instantanée ». *Études de communication*, no 34, p. 93-108.

Lin, Yuwei. 2004. « Hacking Practices and Software Development: A Social Worlds Analysis of ICT Innovation and the Role of Free/Libre Open Source Software ». Thèse de doctorat, York (Royaume-Uni), University of York. En ligne <http://opensource.mit.edu/papers/lin2.pdf>. Consulté le 30 avril 2012.

———. 2005. « Diversity of Knowledge and Dynamics of Knowledge Creation in Floss Communities ». *Working paper v. 0.9 discussed at the FADO seminar Vrije Universiteit Amsterdam*. En ligne <http://www.feweb-vu.nl/dbfilestream.asp?id=2474>. Consulté le 2 septembre 2007.

———. 2006a. « Gender Dilmmas in the Free/Libre Open Source Software Development ». In *Encyclopedia of Gender and Information Technology*. Hershey, PA : Idea Group Inc.

———. 2006b. « Techno-Feminist View on the Open Source Software Development ». *Encyclopedia of Gender And Information Technology*, p. 1148–1154.

Lindtner, Silvia. 2009. « Gaming Codes in China : Cultivating Cool and Socio-Technical Distinction Work ». In *What is Code? What is Coding? Emerging STS Approaches in Studying Computer Code. Society for Social Studies of Science 2009 Conference* (Washington DC, 28 octobre au 1er novembre 2009).

Mackenzie, Adrian. 2005. « The Performativity of Code: Software and Cultures of Circulation ». *Theory Culture Society*, vol. 22, no 1 (1février), p. 71-92.

———. 2006. *Cutting code : software and sociality*. New York : Peter Lang.

Marcus, George M. 1995. « Ethnography in/of the World System: The Emergence of Multi-Sited Ethnograpy ». *Annual Review of Anthropology*, vol. 24, p. 95-117.

Marino, Mark C. 2006. « Critical Code Studies ». *Electronic Book Review*. En ligne <http://www.electronicbookreview.com/thread/electropoetics/codology>. Consulté le 30 avril 2012.

———. 2010. « Critical Code Studies and the *Electronic book Review*: An Introduction ». *Electronic Book Review*. En ligne <http://www.electronicbookreview.com/thread/firstperson/ningislanded>. Consulté le 14 janvier 2011.

Martin, David et John Rooksby. 2006. « Knowledge and Reasoning about Code in a Large Code Base ». *TeamEthno-online Issue*, no 2.

Martin, Emily. 1994. *Flexible Bodies : Tracking Immunity in American Culture from the Days of Polio to the Age of AIDS*. Boston : Beacon Press.

Merton, Robert. 1973. *The Sociology of Science*. Chicago : University of Chicago Press.

Millerand, Florence. 1998. « Usages des NTIC: les approches de la diffusion, de l'innovation et de l'appropriation (1ère partie) ». *COMMposite*, vol. 98, no 1.

———. 2003. « L'appropriation du courrier électronique en tant que technologie cognitive chez les enseignants chercheurs universitaires: vers l'émergence d'une culture numérique? ». Thèse de doctorat, Montréal, Université de Montréal.

Millerand, Florence et Karen S. Baker. 2010. « Who Are the Users? Who Are the Developers? Webs of Users and Developers in the Development Process of a Technical Standard ». *Information Systems Journal*, vol. 20, no 2 (mars), p. 137-161.

Millerand, Florence, Serge Proulx et Julien Rueff (dir. publ.). 2009. *Web social : Mutation de la communication*. Québec : Presses de l'Université du Québec.

Moglen, Eben. 1999. « Anarchism Triumphant : Free Software and the Death of Copyritht ». *First Monday*, vol. 4, no 8.

Mol, Annemarie. 2002. *The Body Multiple: Ontology in Medical Practice*. Durham : Duke University Press.

Morrison, Philip et Phylis Morrison. 1999. « 100 or so Books that shaped a Century of Science ». *American Scientist Online*, vol. 87, no 6. En ligne <http://www.americanscientist.org/bookshelf/pub/100-or-so-books-that-shaped-a-century-of-science>. Consulté le 30 avril 2012.

Nachi, Mohamed. 2006. *Introduction à la sociologie pragmatique*. Paris : Armand Colin.

Naur, Peter. 1963. « The Design of the GIER ALGOL compiler Part I ». *BIT*, vol. 3, no 2 (juin), p. 124-140.

Nguyen, Lilly. 2009. « Development in Practice : Imagining and Designing Free and Open Source Software in Vietnam ». In *What is Code? What is Coding? Emerging STS Approaches in Studying Computer Code. Society for Social Studies of Science 2009 Conference* (Washington DC, 28 octobre au 1er novembre 2009).

Norman, Donald A. 1993. « Les artefacts cognitifs ». In *Les objets dans l'action. De la maison au laboratoire.*, p. 15-34. Paris : EHESS.

O'Reilly, Tim. 2007. « What Is Web 2.0. Design Patterns and Business Models for the Next Generation of Software ». *Communications & Strategies*, no 65, p. 17-37.

Oram, Andy et Greg Wilson. 2007. *Beautiful Code : Leading Programmers Explain How They Think (Theory in Practice)*. Sebastopol (Calif.) : O'Reilly Media, Inc.

Orlikowski, Wanda J. et JoAnne Yates. 1994. « Genre Repertoire: The Structuring of Communicative Practices in Organizations ». *Administrative Science Quarterly*, vol. 39, no 4 (1décembre), p. 541-574.

Oudshoorn, Nelly et Trevor Pinch. 2003. *How Users Matter : The Co-Construction of Users and Technology*. Cambridge (Mass.) : The MIT Press.

Passoth, Jan-Hendrik. 2009. « Coding the Code of Blogging Infrastructure: The Case of RSS ». In *What is Code? What is Coding? Emerging STS Approaches in Studying Computer Code. Society for Social Studies of Science 2009 Conference* (Washington DC, 28 octobre au 1er novembre 2009).

Pène, Sophie. 2005. « Les agencements langagiers de la qualité ». In *Langage et Travail : Communication, cognition, action*, sous la dir. de Anni Borzeix et Béatrice Fraenkel, p. 303-321. 2ième ed. CNRS.

Proulx, Serge. 2002. « L'identité québécoise à l'ère des réseaux numériques : entre cyberespace et mondialisation ». In *Colloque Bogues 2001 : globalisme et pluralisme* (Montréal, 24-27 avril 2002).

———. 2005a. « Penser la conception et l'usage des objets communicationnels. Vers un constructivisme critique ». In *Communication : horizons de pratiques et de recherche*, p. 295-316. Québec : Presse de l'Université du Québec.

———. 2005b. « Penser les usages des TIC aujourd'hui : enjeux, modèles, tendances ». In *Enjeux et usages des TIC : aspects sociaux et culturels*, sous la dir. de Lise Vieira et Nathalie Pinède, p. 7-20. Bordeaux : Presses universitaires de Bordeaux.

———. 2006. « Les militants du code : la construction d'une culture technique alternative ». In *Colloque « logiciel libre en tant que modèle d'innovation sociotechnique », 74e congrès de l'ACFAS* (Université McGill, Montréal, 16 mai 2006). En ligne <http://cmo.uqam.ca/node/62>. Consulté le 8 décembre 2008.

———. 2007a. « Laboratoire de communication médiatisée par ordinateur (LabCMO) un programme de recherche orienté vers les techno-activistes ». *Cahiers du SFSIC, Société française des sciences de l'information et de la communication*, no 1.

———. 2007b. « Web participatif : vers un engagement citoyen de l'usager ? ». In *Conférence Éthique et droits de l'homme dans la société de l'information, Commission française pour l'UNESCO et Conseil de l'Europe* (Strasbourg, 13-14 septembre 2007).

———. 2007c. « Interroger la métaphore d'une société de l'information: Horizon et limites d'une utopie ». *Communication et langages*, no 152, p. 107-124.

———. 2012. « La sociologie de la communication au prisme des études sur la science et la technique ». In *Connexions. Communication numérique et lien social*, sous la dir. de Serge Proulx et Annabelle Klein. Namur (Belgique) : Presses universitaires de Namur.

Proulx, Serge, Stéphane Couture et Julien Rueff (dir. publ.). 2008. *L'action communautaire québécoise à l'ère du numérique*. Québec : Presses de l'Université du Québec.

Proulx, Serge et Anne Goldenberg. 2010. « Internet et la culture de la gratuité ». *Revue du Mauss*, no 35, p. 503–517.

Proulx, Serge, Lorna Heaton, Mary Jane Kwok Choon et Mélanie Millette. 2011. « Paradoxical Empowerment of Produsers in the Context of Informational Capitalism ». *New Review of Hypermedia and Multimedia*, vol. 17, no 1, p. 9-29.

Raymond, Eric S. 2001. *The Cathedral and the Bazaar.* Sebastopol (Calif.) : O'Reilly Media, Inc.

Reagle, Joseph. 2007. « Bug Tracking Systems as Public Spheres ». *Techné : Research in Philosophy and Technology*, vol. 11, no 1. En ligne <http://scholar.lib.vt.edu/ejournals/SPT/v11n1/reagle.html>. Consulté le 25 février 2011.

Richardson, Joanne. 2002. « Logiciel libre et éthique du développement de soi ». *Multitudes*, vol. 1, no 8, p. 188-199.

Roberts, J. A, I. Hann et S. A Slaughter. 2006. « Understanding the Motivations, Participation, and Performance of Open Source Software Developers: A Longitudinal Study of the Apache Projects ». *Management Science*, vol. 52, no 7, p. 984-999.

Robles, Gregorio et Jesus M. Gonzalez-Barahona. 2004. « Executable Source Code and Non-Executable Source Code: Analysis and Relationships ». In *Proceedings of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 04)*, p. 149-147.

Rogers, Richard. 2009. *The End of the Virtual : Digital Methods.* Amsterdam : Amsterdam University Press.

Rooksby, John et David Martin. 2006. « Ethnographies of Code ». *TeamEthno-online*, no 2, p. 1-2.

Sack, Warren, Françoise Détienne, Nicolas Ducheneaut, Jean-Marie Burkhardt, Dilan Mahendran et Flore Barcellini. 2006. « A Methodological Framework for Socio-Cognitive Analyses of Collaborative Design of Open Source Software ». *Computer Supported Cooperative Work (CSCW)*, vol. 15, no 2, p. 229-250.

Salin, Peter. 1991. « Freedom of Speech in Software ». *PhilSalin.com*. En ligne <http://www.philsalin. com/patents.html>. Consulté le 30 avril 2012.

Scacchi, W. 2007. « Free/open source software development: Recent research results and methods ». *Advances in Computers*, vol. 69, p. 243–295.

Searle, John R. 1976. « A Classification of Illocutionary Acts ». *Language in Society*, vol. 5, no 1, p. 1-23.

Shapin, Steven et Simon Schaffer. 1985. *Leviathan and the Air Pump : Hobbes, Boyle, and the Experimental Life*. Princeton (N.J.) : Princeton University Press.

Shukla, Shilpa V. et David F. Redmiles. 1996. « Collaborative Learning in a Software Bug-Tracking Scenario ». In *Conference on Computer Supported Cooperative Work (CSCW 96)*.

Simondon, Gilbert. 1958. *Du mode d'existence des objets techniques*. Paris : Aubier-Montaigne.

———. 2001. *Du mode d'existence des objets techniques*. 4e éd. rev. et augm. Paris : Aubier-Montaigne.

———. 2007. *L'individuation psychique et collective : à la lumière des notions de forme, information, potentiel et métastabilité / Gilbert Simondon*. Paris : Aubier.

Sokal, Alan et Jean Bricmont. 1997. *Impostures intellectuelles*. Paris : Odille Jacob.

SPIP. 2002. « L'histoire minuscule et anecdotique de SPIP - SPIP ». En ligne <http://www.spip.net/fr_article918.html>. Consulté le 25 février 2011.

Stallman, Richard. 2002. « La passion du libre entretien avec Richard Stallman (entretien réalisé par Jérôme Gleizes et Aris Papathéodou) ». *Samizdat | Biblioweb*. En ligne <http://biblioweb.samizdat.net/article18.html>. Consulté le 30 avril 2012.

———. 2004. « The GNU Operating System and the Free Software Movement ». In *Open Sources Voices from the Open Source Revolution*. Sebastopol (Ca) : O'Reilly Media, Inc. En ligne <http://www.oreilly.com/catalog/opensources/book/stallman.html>. Consulté le 23 novembre 2011.

Star, S. L. et J. R. Griesemer. 1989. « Institutional Ecology, « Translations » and Boundary Objects: Amateurs and Professionals in Berkeley's Museum of Vertebrate Zoology, 1907-1939 ». *Social Studies of Science*, vol. 19, no 3, p. 387-420.

Star, Susan Leigh. 1995a. « Introduction ». In *Ecologies of Knowledge: Work and Politics in Science and Technology*, p. 1-35. Albany (NY) : State University of New York Press.

———. 1995b. « The politics of Formal Representations. Wizards, Gurus, and Organizational Complexity ». In *Ecologies of Knowledge. Work and Politics in Science and Technology*, p. 88-118. Albany (NY) : State University of New York Press.

———. 1995c. *Ecologies of Knowledge : Work and Politics in Science and Technology*. State University of New York Press.

———. 1999. « The Ethnography of Infrastructure ». *American Behavioral Scientist*, vol. 43, no 3, p. 377-391.

Star, Susan Leigh et Karen Ruhleder. 1996. « Steps Toward an Ecology of Infrastructure: Design and Access for Large Information Spaces ». *Information Systems Research*, vol. 7, no 1, p. 111 - 134.

Star, Susan Leigh et Anselm Strauss. 1999. « Layers of Silence, Arenas of Voice : The Ecology of Visible and Invisible Work ». *Computer Supported Cooperative Work*, vol. 8, no 1-2, p. 9-30.

Stiegler, Bernard, Alain Giffard et Christian Faure. 2009. *Pour en finir avec la mécroissance : quelques réflexions d'Ars Industrialis*. Paris : Flammarion.

Suchman, Lucy. 1987. *Plans and Situated Actions : The Problem of Human-Machine*. New York : Cambridge University Press.

———. 1999. « Critical Practices ». *Anthropology of Work Review*, vol. 20, no 1, p. 12-14.

———. 2006. « Agencies in Technology Design: Feminist Reconfigurations ». En ligne <http://www.lancs.ac.uk/fss/sociology/papers/suchman-agenciestechnodesign.pdf>.

———. 2007. *Human-Machine Reconfigurations : Plans and Situated Actions*. 2ième éd. Cambridge ; New York : Cambridge University Press.

———. 2008. « Feminist STS and the Sciences of the Artificial ». In *The Handbook of Science and Technology Studies*, sous la dir. de Edward J. Hackett, Olga Amsterdamska, Michael Lynch et Judy Wajcman, p. 139-164. 3ième édition. Cambridge : MIT Press.

Taylor, J.R. et E.J. Van Every. 2000. *The Emergent Organization : Communication As Its Site and Surface*. Mahwah : Lawrence Erlbaum Associates.

Turing, Alan M. 1936. « On Computable Numbers, with an Application to the Entscheidungsproblem ». *Proceedings of the Mathematical Society*, vol. 42, no 2, p. 230-265.

Vaucelle, Alain et Henri Hudrisier. 2010. « Langages structurés et lien social ». *tic & société*, vol. 4, no 1. En ligne <http://ticetsociete.revues.org/790>. Consulté le 7 janvier 2011.

Wajcman, Judy. 2004. *TechnoFeminism*. Cambridge, UK ; Malden, MA : Polity Press.

Weill, Claire. 2007. « *Les trois écritures* de Clarisse Herrenschmidt ». *Transversales Sciences & culture*. En ligne <http://grit-transversales.org/article.php3?id_article=191>. Consulté le 7 janvier 2011.

Winner, Langdon. 1980. « Do Artifacts Have Politics? ». *Daedalus*, vol. 109, no 1, p. 121-136.

———. 2002. « Les artefacts font-ils de la politique? ». In *La baleine et le réacteur. À la recherche de limites au temps de la haute technologie*, p. 35-74. Paris : Descartes et cie.

Woolgar, Steve. 1991. « Configuring the User: The Case of Usability Trials. ». In *A Sociology of Monsters: Essays on Power, Technology, and Domination*, sous la dir. de John Law, p. 57-99. London : Routledge.

Wortham, Jenna. 2012. « A Surge in Learning the Language of the Internet ». *The New York Times*, 27mars. En ligne <http://www.nytimes.com/2012/03/28/technology/for-an-edge-on-the-internet-computer-code-gains-a-following.html>. Consulté le 28 mars 2012.

APPENDICE A

PROFIL DES PARTICIPANTS AUX ENTREVUES

Entrevues SPIP

ID	Date	Durée de l'entrevue	Sexe	Âge	Rôle dans la communauté
spip01	12 juillet 09	178 min	h	39	Développeur/Membre du core
spip02	21 août 09	65min	h	~40	Utilisateur de SPIP
spip03	22 août 09	96 min	h	~50	Développeur/Membre du core
spip04	27 août 09	64 min	h	19	Utilisateur de SPIP/Création de squelettes
spip05	24 février 10	89 min	h	52	Utilisateur de SPIP
~~spip06~~	~~20 mars 10~~	~~135 min~~	~~h~~	~~~50~~	~~Utilisateur de SPIP~~
spip07	30 mars 10	111 min	h	60	Utilisateur de SPIP
spip08	11 avril 10	91 min	f	35	Utilisatrice/membre du core team
spip09	10 avril 10	151 min	h	~30	Développeur de plugins, administrateur d'un serveur
spip10	11 avril 10	92 min	h	28	Développeur de plugins
spip11	15 mai 10	102 min	h	37	Développeur/membre du core

Entrevues symfony

ID	Date	Durée de l'entrevue	Sexe	Âge	Rôle dans la communauté
sf01	12 juin 09	40min	h	~30	Développeur de plugin
sf02	23 juin 09	49 min	h	35	Membre du core
sf03	10 juillet 09	81 min	h	35	Ancien membre du core
sf04	16 juillet 09	84 min	h	~35	Développeur de plugin Utilisateur
sf05	9 mars 10	131 min	f	~25	Formatrice
sf06	12 mars 10	97 min	h	24	Écriture du plugin. Soumission de patche
sf07	3 avril 10	77 min	h	22	Formateur. Écriture de plugin
sf08	16 avril 10	87 min	h	~30	Ancien membre du core
sf09	28 avril 10	76 min	h	28	Développeur de plugin
sf10	29 avril 2010	61 min	h	44[204]	« fédérer et monter une communauté de développeur »

204 A indiqué être né en 1967.

APPENDICE B

EXEMPLE DE GRILLE D'ENTREVUE

Entrevue sf06

Thème	Sous-thème	Questions d'entrevues	Commentaires
I - Introduction		-Présentation de la recherche -Consentement -Format de l'entrevue (entrevue semi-dirigée) -Quel âge ?	
II - Projet	Implication dans le projet	-Pourrais me décrire ta contribution dans le projet symfony ? -Comment en es-tu arrivée à t'impliquer dans Symfony ? -Tu n'es pas sur OHLOH ?	Patch -écriture de plugin : (solr).
	Définition du projet	-Qu'est-ce que symfony ?	
	Femmes	-Que penses-tu de la faible participation des femmes dans Symfony ?	
III - Définition et organisation du code	Qu'est-ce que le code ?	-Ma recherche porte précisément sur le « code source ». D'après tes connaissances, est-ce que tu pourrais me définir le « code source » ? -Dans le contexte de symfony, qu'est-ce que le « code source », quelle forme prend-il ?	
IV - Appréciation du code	Lisibilité du code	La lisibilité du code : « Using symfony is so natural and easy for people used to PHP and the design patterns of Internet applications that the learning curve is reduced to less than a day. The **clean design** and **code readability** will keep your delays short. » http://www.symfony-project.org/about -Que signifie « code readibility » ? Qu'est-ce qui favorise selon toi la lisibilité du code, dans Symfony ?	
	Code **explicite**	################## (extrait d'une intervention faite sur un site web)	Particulier au répondant

Thème	Sous-thème	Questions d'entrevues	Commentaires
	Beauté	-Quelques fois, on entend parler de beauté du code, qu'est-ce que tu en penses ? -Est-ce que tu penses que PHP est un beau code ?	
	Langue du code	-Le code de symfony est en anglais, ce qui est quand même particulier. Est-ce que tu trouves ça bien, ou crois-tu plutôt que le code devrait être en anglais ?	
V- Organisation et interaction avec le code	Normes	http://trac.symfony-project.org/wiki/HowToContributeToSymfony#CodingStandards -Qu'est-ce qu'un standard de programmation ? À quoi servent-ils ? -Comment les reconnaître, comment les renforcer ? **Bonnes pratiques**	
	Comment est organisé le code source ?	-Dans le cadre du projet SPIP, peux-tu me décrire comment est organisé le code source ? -On a parlé des différents espaces de discussions pour SPIP. Pourrais-tu m'énumérer les espaces où l'on retrouve du code ?	
	Comment organises-tu ton code, comment écris-tu le code ?	-Quel est ta relation avec le code ? -Comment organises-tu ton code, -Comment écris-tu le code ? -Quel logiciel utilises-tu pour écrire du code ?	
	Code privé et code public.	-Y a-t-il du code que tu gardes chez toi, en privé? Pourquoi ? -Sur quels critères te base-tu pour rendre public ton code ? -Que faut-il faire pour rendre le code public ?	
VI - Finalisation		-Grandes discussions dans Symfony -Connais-tu des personnes qu'il serait intéressant de rencontrer ? -Comment as-tu trouver l'entrevue ?	

APPENDICE C

FORMULAIRE DE CONSENTEMENT

UQÀM

TELECOM ParisTech

FORMULAIRE DE CONSENTEMENT POUR ENTRETIEN

« Le code source informatique comme artefact des assemblages sociotechniques »
Recherche de doctorat de Stéphane Couture
Université du Québec à Montréal
Télécom ParisTech

1. Description du projet
Vous êtes invité(e) à prendre part à une recherche de doctorat en sciences sociales. Ce projet s'intéresse aux pratiques collectives de programmation informatique réalisées sur Internet, en particulier dans le cadre de développement de logiciels libres et « open source ». Il s'inscrit dans une démarche sociologique concernée par les formes de partage d'informations qui se développent aujourd'hui, en particulier sous l'appellation web 2.0. Son objectif est plus spécifiquement de comprendre, d'un point de vue sociologique, le rôle du « code source », comme résultat et source d'engagement dans ces pratiques collaboratives.

Ce projet de thèse est réalisé par Stéphane Couture conjointement à l'Université du Québec à Montréal (UQAM) et à Télécom ParisTech, à Paris. Il est réalisé sous la supervision du professeur Serge Proulx à l'UQAM et du professeur Christian Licoppe à Télécom ParisTech. Il reçoit l'appui financier du Conseil de recherche en sciences humaines du Canada (CRSH).

2. Votre participation
Votre participation à cette recherche consiste en une entrevue individuelle au cours de laquelle il vous sera demandé de commenter votre activité d'écriture de code source et votre expérience dans un projet collectif de développement de logiciel. Cette entrevue sera enregistrée numériquement avec votre permission et prendra environ une heure de votre temps. Le lieu et l'heure de l'entrevue sont à convenir avec le chercheur.

Il n'y a pas de risque d'inconfort important associé à votre participation à cette recherche. Cependant, vous demeurez libre de ne pas répondre à une question que vous estimez embarrassante sans avoir à vous justifier. Il est également entendu que vous puissiez décider de suspendre ou de mettre fin à l'entrevue pour quelque raison que ce soit. Dans ce cas, l'enregistrement numérique sera immédiatement détruit et votre intervention ne sera pas retenue pour la recherche.

3. Confidentialité
Les renseignements recueillis lors de l'entrevue sont confidentiels et seul le chercheur aura accès à votre enregistrement et au contenu intégral de sa transcription. Aucune donnée qui permettrait de vous identifier ne sera dévoilée lors de l'analyse et de la présentation écrite ou orale des résultats de la recherche. Un pseudonyme vous désignera dans tous les rapports, articles, conférences, présentations, etc. Le présent formulaire sera conservé sous clé et les documents numériques (enregistrement numérique et transcription) seront conservés dans un dossier crypté (avec protocole AES-256) pour la durée totale du projet. Tous ces documents seront détruits deux ans après la publication de la thèse de doctorat.

4. Questions et droits relatifs à votre participation
Vous pouvez contacter par courriel le chercheur responsable de la recherche (steph@stephcouture.info) ou ses superviseurs pour des questions additionnelles sur le projet ou sur vos droits en tant que participant à cette recherche. Coordonnées des superviseurs :
Serge Proulx : proulx.serge@uqam.ca, (514) 987-3000 poste: 4533
Christian Licoppe (christian.licoppe@telecom-paristech.fr). 33 (0)1 45 81 81 16

Le Comité institutionnel d'éthique de la recherche avec des êtres humains de l'UQAM a approuvé ce projet de recherche. Pour des informations concernant les responsabilités du chercheur sur le plan de l'éthique de la recherche ou pour formuler une plainte ou des commentaires, vous pouvez contacter le Président du Comité institutionnel d'éthique de la recherche de l'UQAM Joseph Josy Lévy : levy.joseph_josy@uqam.ca, (514) 987-3000 poste 4483 ou poste 7753 (secrétariat du comité).

5. Remerciements
Votre collaboration est essentielle pour la réalisation de ce projet et nous tenons à vous en remercier. Si vous souhaitez réviser le verbatim de cet entretien ou obtenir un résumé écrit des principaux résultats de cette recherche, veuillez ajouter vos coordonnées plus bas.

6. SIGNATURES

Je _____ reconnais avoir lu le présent formulaire de consentement et consens volontairement à participer à ce projet de recherche. Je reconnais aussi que l'intervieweur a répondu à mes questions de manière satisfaisante et que j'ai disposé de suffisamment de temps pour réfléchir à ma décision de participer. Je comprends que ma participation à cette recherche est totalement volontaire et que je peux y mettre fin en tout temps, sans pénalité d'aucune forme, ni justification à donner. Il me suffit d'en informer le responsable du projet.

___ Je veux réviser le verbatim de l'entretien

___ Je veux recevoir un résumé des résultats de la recherche

X_____
Signature du participant/participante à la recherche Date :

Coordonnées (pour réviser le verbatim et/ou recevoir les résultats) :

X_____
Signature du chercheur (S. Couture) Date :

Veuillez conserver le premier exemplaire de ce formulaire de consentement pour communication éventuelle avec l'équipe de recherche et remettre le second au chercheur.

APPENDICE D

CHARTE DE FONCTIONNEMENT DE LA ZONE DE SPIP

CHARTE (de fonctionnement de la zone)[205]

Droit d'auteur ¶
En accord avec le droit français, toute contribution faite sur zone.spip.org, de quelque nature qu'elle soit, reste en tant que telle la propriété de son auteur, mais se fond dans l'« œuvre collective » qu'est le projet SPIP. Toutes les contributions ajoutées sur ce site doivent être « libres » au sens de GNU (licence GPL, LGPL, FDL sans sections invariantes, licence Art Libre...), de manière à pouvoir s'intégrer dans le projet global SPIP. Voir la liste des licences acceptées pour une inclusion de votre projet dans la zone spip. En particulier, si la contribution porte sur l'adaptation d'une œuvre antérieure, il convient de s'assurer des conditions de licence de ladite œuvre (pas question d'accepter une autorisation restreinte à ce seul site, par exemple). Si vous souhaitez travailler avec trac sur un projet ne respectant pas ces critères, il faut le faire sur un autre serveur.

Respect des buts et valeurs du projet SPIP ¶
Rappelons que SPIP est un logiciel libre, et chaque personne qui l'utilise peut en faire ce qu'elle veut ; toutefois la participation à la SPIP Zone doit être faite dans le cadre des buts et valeurs promus par le projet initial du minirézo, et notamment :

- promouvoir et défendre la liberté d'expression de tous sur Internet
- une défiance vis-à-vis de l'argent
- le respect de l'identité de chacun

Cela implique, entre autres, un effort pour **internationaliser** ses contributions, veiller à ce que le langage et le fonctionnement choisis soient non-sexistes, une priorité accordée aux besoins **associatifs** sur les besoins marchands, etc.
Ce site n'est pas une plateforme de développement pour des versions militaires ou *business-oriented* de SPIP qui viendraient en changer la nature. Il n'a pas non plus vocation à servir de support de communication ou de publicité pour consultants. Si vous pensez que votre activité professionnelle peut entrer en conflit avec ces exigences, on peut en parler et trouver des solutions (l'usage d'un pseudo par exemple).

Bon fonctionnement coopératif ¶
Un tel projet collectif ne peut fonctionner que si chacun fait un effort pour ne pas empiéter sur les projets d'autrui ; il faut aussi veiller, d'un autre côté, à ce que personne ne se déclare « responsable » d'un projet puis le laisse mourir faute d'attention. Au-delà des règles habituelles qui permettent de ne pas se marcher sur les pieds (communiquer, prévenir les autres, etc.), cela passe par une bonne compréhension des méthodes générales du développement de SPIP et de ses contributions. Ce n'est pas chose facile, et en cas de problème il est important de limiter le recours à la mauvaise foi :-) Il est en particulier obligatoire de respecter les termes des **règles d'intervention** définies dans les projets sur lesquels on intervient.

Mécanisme de résolution des conflits ¶
Un règlement interne ne saurait être complet sans préciser ses mécanismes de résolution des conflits. Cette section évoluera probablement au cours du temps, en fonction du nombre et de la nature des conflits à traiter, mais il faut bien démarrer par quelque chose.
Une commission d'arbitrage sera désignée. En cas de conflit, tout participant peut saisir cette commission. Celle-ci statue en conscience, en se basant sur les critères définis ci-dessus, et dispose de toute latitude dans le règlement du conflit.

---

205 http://zone.spip.org/trac/spip-zone/wiki/CharteDeFonctionnement (consulté le 8 novembre 2011).

APPENDICE E

LE TICKET #4152 DE SYMFONY

symfony

Development
#4152 (Confirm popup make javascript call don't work with IE)

You must first sign up to be able to contribute.

Ticket #4152 (closed defect: fixed)

Confirm popup make javascript call don't work with IE			Opened 3 years ago Last modified 2 years ago
Reported by:	Leonard	Assigned to:	FabienLange
Priority:	major	Milestone:	1.2.9
Component:	helpers	Version:	1.1.0
Keywords:	link_to_function, javascript, IE, confirm, popup	Cc:	
Qualification:	Ready for core team		

Description

While using a remote_function, link_to_function, link_to remote or any function that uses a 'confirm' option, the call don't work on IE6 and IE7

the generated code is "... onClick="if(confirm('Confirmation message'){..." whereas it should be "... onClick="if(window.confirm('Confirmation message'){..."

Attachments

- window_confirm.diff (0.6 kB) - added by intru on 07/10/09 15:56:48.

Change History

08/07/08 17:34:06 changed by Leonard (in reply to: ↑ description)

- status changed from new to closed .
- resolution set to fixed .

Replying to Leonard:

> While using a remote_function, link_to_function, link_to remote or any function that uses a 'confirm' option, the call don't work on IE6 and IE7 the generated code is
>
> "... onClick="if(confirm('Confirmation message'){..."
>
> whereas it should be
>
> "... onClick="if(window.confirm('Confirmation message'){..."

PATCH

File : php5\PEAR\symfony\helper\JavascriptHelper?.php

line 94 is :

```
$html_options['onclick'] = "if(confirm('$confirm')){ $function;); return false;";
```

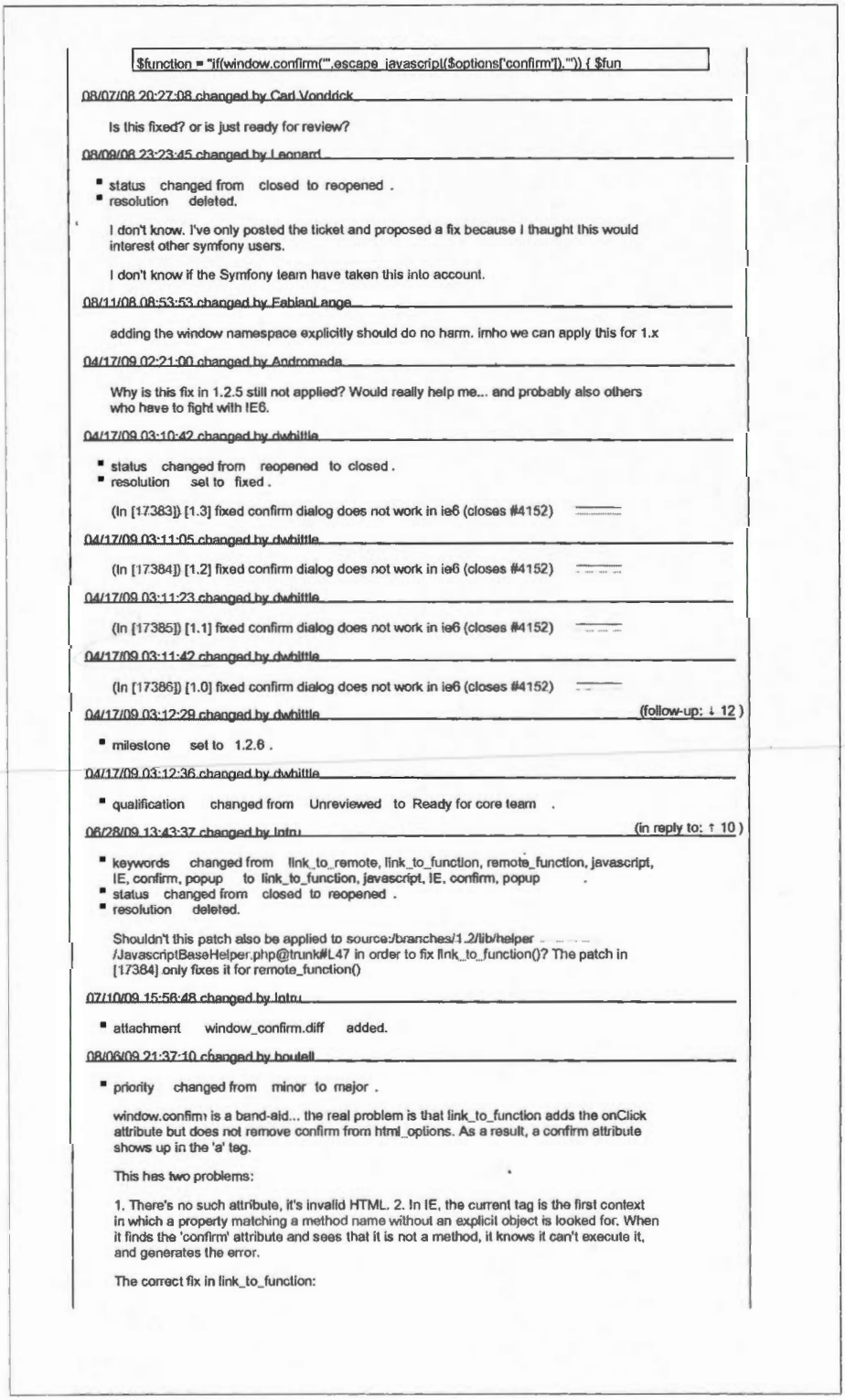
line 94 should be :

```
$html_options['onclick'] = "if(window.confirm('$confirm')){ $function;); return fals
```

line 482 is :

```
$function = "if(confirm('".escape_javascript($options['confirm'])."')) { $function;
```

line 482 should be :



$function = "if(window.confirm('".escape_javascript($options['confirm'])."')) { $fun

08/07/08 20:27:08 changed by Carl Vondrick

Is this fixed? or is just ready for review?

08/09/08 23:23:45 changed by Leonard

- status changed from closed to reopened .
- resolution deleted.

I don't know. I've only posted the ticket and proposed a fix because I thaught this would interest other symfony users.

I don't know if the Symfony team have taken this into account.

08/11/08 08:53:53 changed by FabianLange

adding the window namespace explicitly should do no harm. imho we can apply this for 1.x

04/17/09 02:21:00 changed by Andromeda

Why is this fix in 1.2.5 still not applied? Would really help me... and probably also others who have to fight with IE6.

04/17/09 03:10:42 changed by dwhittle

- status changed from reopened to closed .
- resolution set to fixed .

(In [17383]) [1.3] fixed confirm dialog does not work in ie6 (closes #4152)

04/17/09 03:11:05 changed by dwhittle

(In [17384]) [1.2] fixed confirm dialog does not work in ie6 (closes #4152)

04/17/09 03:11:23 changed by dwhittle

(In [17385]) [1.1] fixed confirm dialog does not work in ie6 (closes #4152)

04/17/09 03:11:42 changed by dwhittle

(In [17386]) [1.0] fixed confirm dialog does not work in ie6 (closes #4152)

04/17/09 03:12:29 changed by dwhittle (follow-up: ↓ 12)

- milestone set to 1.2.6 .

04/17/09 03:12:36 changed by dwhittle

- qualification changed from Unreviewed to Ready for core team .

06/28/09 13:43:37 changed by lotru (in reply to: ↑ 10)

- keywords changed from link_to_remote, link_to_function, remote_function, javascript, IE, confirm, popup to link_to_function, javascript, IE, confirm, popup .
- status changed from closed to reopened .
- resolution deleted.

Shouldn't this patch also be applied to source:/branches/1.2/lib/helper /JavascriptBaseHelper.php@trunk#L47 in order to fix link_to_function()? The patch in [17384] only fixes it for remote_function()

07/10/09 15:56:48 changed by lotru

- attachment window_confirm.diff added.

08/06/09 21:37:10 changed by houtell

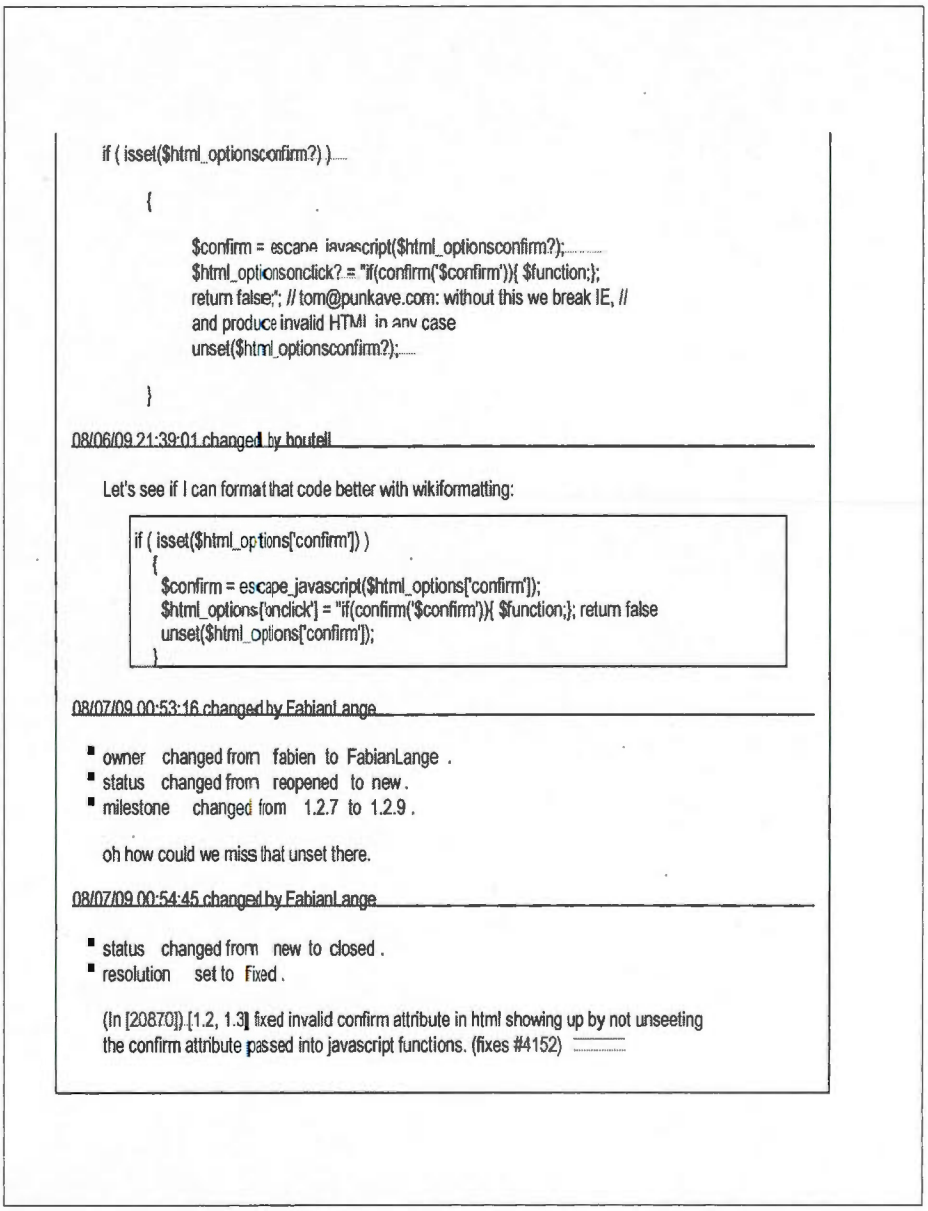
- priority changed from minor to major .

window.confirm is a band-aid... the real problem is that link_to_function adds the onClick attribute but does not remove confirm from html_options. As a result, a confirm attribute shows up in the 'a' tag.

This has two problems:

1. There's no such attribute, it's invalid HTML. 2. In IE, the current tag is the first context in which a property matching a method name without an explicit object is looked for. When it finds the 'confirm' attribute and sees that it is not a method, it knows it can't execute it, and generates the error.

The correct fix in link_to_function:



```
if ( isset($html_optionsconfirm?) )
    {
        $confirm = escape_javascript($html_optionsconfirm?);
        $html_optionsonclick? = "if(confirm('$confirm')){ $function;};
        return false;"; // tom@punkave.com: without this we break IE, //
        and produce invalid HTML in any case
        unset($html_optionsconfirm?);
    }
```

08/06/09 21:39:01 changed by boutell

Let's see if I can format that code better with wikiformatting:

```
if ( isset($html_options['confirm']) )
  {
    $confirm = escape_javascript($html_options['confirm']);
    $html_options['onclick'] = "if(confirm('$confirm')){ $function;}; return false
    unset($html_options['confirm']);
  }
```

08/07/09 00:53:16 changed by FabianLange

- owner changed from fabien to FabianLange .
- status changed from reopened to new .
- milestone changed from 1.2.7 to 1.2.9 .

oh how could we miss that unset there.

08/07/09 00:54:45 changed by FabianLange

- status changed from new to closed .
- resolution set to Fixed .

(In [20870]) [1.2, 1.3] fixed invalid confirm attribute in html showing up by not unseeting the confirm attribute passed into javascript functions. (fixes #4152)

Page du ticket :
http://trac.symfony-project.org/ticket/4152 (consulté le 17 novembre 2011)