International Journal of Communication 11(2017), 1112–1132 1932–8036/20170005

Values and Configuration of Users in the Design of Software Source Code

STÉPHANE COUTURE[1]
Glendon College, York University, Canada

Based on an empirical study of two free and open source software projects, this article focuses on how choices of design within software source code are articulated with values and may favor certain categories of actors over others. After introducing the aim and approach of the study and presenting the studied projects, the article analyzes two controversies that show the articulation of values, configuration of users, and source code design. In conclusion, I argue for the importance for communication and media studies to study the use of source code and how its design may reflect values or may facilitate or constrain the agencies of certain categories of people.

Keywords: software source code, free and open source software, digital technologies, configuration of users, values, science and technology studies

When I am a graphic designer, my interface is Photoshop, with its buttons, its windows, etc. When I am developer, my interface is code. It is through code that I interact with what I am building, a program. (Interview sf03, July 2009)

This quote from an interview I conducted in the course of this study grasps a fundamental argument I want to bring forward in this article: that software source code should be analyzed as an interface with which actors interact to build or modify software. Although metaphors of code are regularly mobilized in communication studies, few studies seem to have been done to closely and empirically investigate what source code exactly is and how its design may reflect values or may facilitate or restrain the capacity of some people to participate in the making of digital technologies. As digital technologies take an always increasing importance in public and personal communication, it is important for communication and media studies to look at how people interact with these artifacts to reconfigure, rebuild, and reassemble them to engage more creatively and responsibly with the computational possibilities they afford (Suchman, 2007).

Stéphane Couture: scouture@glendon.yorku.ca
Date submitted: 2015–05–21

[1] This research received funding from the Social Sciences and Humanities Research Council (SSHRC), and the Fonds de recherche du Québec–Société et culture (FRQSC). Many thanks to Gabriella Coleman, Alessandro Delfanti, Guillaume Latzko-Toth, and members of the Bits bytes and bots group at McGill for reviewing and commenting this work, as well as to Geneviève Szczepanik for her editing work, many comments and general support.

While the argument that "source code matters" may already be well accepted within communication studies, I would insist that we now need to take another step by producing empirical investigations about *how* exactly it matters. The main contribution of this article is to offer a methodological example of how to study source code, from a communicational and social science perspective, and specifically, to address the issues of values within source code. Based on the ethnographic study of two free and open source software projects, the article looks at values and configuration of users in the design of source code. By design, I refer to the ways that software source code is written and organized through, for instance, choices of programming languages and formats, naming conventions, and the general architecture of the software program.

The article is structured as follows: In the next section, I provide a literature review of some work done (or not) on subjects such as source code, code, and software in communication studies and peripheral fields. I then introduce two free and open source projects I studied, expose my methodology, and continue with a first general analysis of coding style and source code organization in each of the studied projects. The next section is the heart of the article and addresses two debates or controversies (one for each of the projects) that look at the articulation of values, configurations of users, and source code design. In the final section, I come back to the idea of source code as an interface and explore an assumption made throughout this article about the use of source code.

Studying Source Code in Communication Studies and Social Science

Software source code—or simply *source code* in the context of this article—is the object of computer programming. It has been defined by Krysia and Grzesiek (2008) as human-readable computer commands that formally specify the operation of software, or more broadly, as "the preferred form of a work for making modifications to it" (Free Software Foundation, 2007, "1. Source Code," para. 1). Although source code is an essential "material" with which digital technologies and artifacts are built, assembled, and reconfigured, still very little work has been done in communication studies and peripheral fields to empirically study this object.[2]

It is, however, true that the notion of *code* has a resonance in communication studies, especially to apprehend the regulatory and prescriptive power of technical objects, platforms, and algorithms. Often cited is the argument put forward by Lessig (2000) that "code is law," to insist on the ways that our behaviors are, at least in part, regulated by the codified architecture of the Internet. Although this thesis is surely highly politically relevant, the analytical tools to empirically apprehend exactly what this code is that is acting as law are much weaker. When Lessig refers to "code writers," for instance, he refers sometimes to individual programmers and at other times to institutions like the FBI or the government. What mostly interests Lessig is the *running code* of the Internet—the set of instructions, usually taking the form of a text, with which programmers interact to modify software—rather than the *source code*.

[2] For instance, at the time of writing (May 26, 2016), a quick search of *source code* on the *International Journal of Communication* website did not yield any results compared with a search of a specific platform—Facebook—which yielded 21 results.

In media studies, the subfields of software studies and code studies often propose to look at the cultural and ideological dimensions of software and code from a literary perspective (Chun, 2008; Fuller, 2008; Galloway, 2006; Marino, 2010). Using a literary and hermeneutic approach, Marino (2010) proposes what he calls "critical code studies" to interpret significant snippets of source code. Although these studies have convincingly demonstrated that source code may be apprehended as a cultural or ideological artifact, their work is generally not tied to thick empirical studies, but tends to be highly interpretative, sometimes focusing on a single culturally significant line of code (Montfort et al., 2012), rather than on complex software systems. The analysis I propose is significantly distinct from these reflections by its sociological rather than literary nature and by the fact that it addresses real-world software.

Numerous studies have also addressed software and code through the study of free and open source software (Coleman, 2012; Demazière, Horn & Zune, 2007; Kelty, 2008). Coleman (2009) shows, for instance, how actors of free and open source software promote a discourse that considers software source code as a form of speech that should be defended through free speech regulations. Although these empirical studies on free and open source software adequately address the values of actors or the way they self-regulate, source code itself is often overlooked. For instance, Coleman notes that ethical values are expressed through many artifacts such as governance charters, and that aesthetic values such as playfulness, pranking, and cleverness are viewed as the core of certain programming languages. However, she does not analyze the specific forms that source code takes in a particular project. In their ethnography of the SPIP project (one of the cases I also studied, which is addressed in this article), Demazière et al. (2007) provide a compelling description of the "modes of regulations" of this community. However, an analysis of the ways in which *source code* itself may participate or echo these modes of regulation is completely absent from their study.

My analysis is closer to some work that has been done in the fields of human–computer interaction (HCI) and computer-supported cooperative work on programming language usability. In the 1990s, Nardi (1993) published a study on "end user programming" in which she analyzes programming devices such as spreadsheets that she considers to be better adapted to the needs of end users. Souza, Froehlich, and Dourish (2005) consider software source code to be a social and technical artifact, and they developed a visualization tool to analyze how social networks are inscribed within the organization of source code. More recently, and echoing my introductory statement, Myers, Ko, LaToza, and Yoon (2016) state that "software development languages and environments are the user interfaces through which developers interact with computers" (p. 44). In their article titled "Programmers Are Users Too," the authors note that the key observation behind their work—that software developers are human—is still surprisingly controversial among researchers.

While these later works address source code usability and echo the fundamental argument I want to make, they are usually done within the computer science side of HCI, and thus are distant from the discipline of communication studies. Most important, my analysis differs from these works by the way it looks more closely at the way values and political cultures are reflected in the design of computer source code. Methodologically, this analysis is inspired by work in science and technology studies and values in design that looks empirically at the way technologies can integrate cultural or moral values such a freedom, privacy, and security (Knobel & Bowker, 2011; Nissembaum, 2005; Shilton, 2013) or can facilitate the inclusion or exclusion of some people. Again, while these studies address different aspects of technology, from algorithms to graphical interfaces, few so far have focused on the design of software source code. This article will address these questions: What are the values and cultures of the different projects? How are these values and cultures eventually inscribed in software source code? How are visions of potential users discussed in the projects and eventually inscribed in source code?[3]

A Comparative Study of Two Free/Open Source Software Projects

The analysis I propose is based on an ethnographic study I conducted in 2009 and 2010 in France. The study analyzed two free/open source software projects written in the PHP programming language: SPIP and Symfony, both of which function as engines for the operation of websites.

Initiated in 2001 by a group of French activists, SPIP is described as "a publishing system for the Internet in which great importance is attached to collaborative working, to multilingual environments, and to simplicity of use for web authors" (SPIP, n.d., para. 1). The community is primarily made up of French nationals, and organizational communication occurs in French. SPIP is mostly aimed at nonprofit and small associations as well as alternative media, but it has also found use in a few government institutions. An e-mail sent to one of the project's mailing lists in 2010 indicated that there were 11,604 websites running SPIP. While the community is still active more than 10 years after its creation, SPIP is sometimes criticized as "outdated software" (Libre-Fan, 2008). This is, naturally, disputed by many of those involved in SPIP who nevertheless agree that SPIP is distinctive from many other PHP projects, in terms of both values and practices.[4]

The Symfony software, at the time of my study, was described as a "full-stack framework, a library of cohesive classes written in PHP5" (Symfony, n.d., para. 2). Instead of a single software, Symfony is best seen as a *framework*—that is, as a set of separate software components (called *libraries* in the programming vocabulary)—that can be assembled to build Web applications and complex websites. While the development of Symfony relies on a very active international community, it was initiated in France, where its core membership remains. Discussion generally occurs in English. Unlike SPIP, an independent project, Symfony is sponsored by a private company called Sensio Lab. Indeed, the software started as an internal project named Sensio Framework and, for this reason (as I will describe later), many components of the source code are prefixed with the letters *sf*. When its developers decided to open source the project, the Sensio Framework was renamed in a way which would retain the existing

[3] Work on values in design tends to focus on the inscription or embedding of values, rather than on formal definitions. For instance, Friedman, Kahn, and Borning (2008) define values very broadly as "what a person or group of people consider important in life" (p. 70). Going along this line, my goal is to study articulations between values, visions of users, and technological design, rather than to engage in a conceptual debate on values.

[4] As I previously mentioned, the SPIP project has been the object of an ethnographic study by Demazière et al. (2007) that focused mainly on the regulation of the community. My analysis differs from theirs by the focus I give to source code and its design.

prefixes—thus, the name *S*ym*f*ony.

A first set of distinctions between the project is of a more sociological nature and concerns the national origins and professional backgrounds of people involved in these projects. SPIP, being a predominately French-speaking project, has trouble reaching a public outside of France. In contrast, Symfony assembles actors from various countries, such as France, the United States, Germany, and the Netherlands.[5] More striking are the differences regarding professional and academic backgrounds. In Symfony, the majority of actors that I met completed an undergraduate degree in computer science, engineering, or multimedia. In SPIP, the situation is much more eclectic. While some of the actors I interviewed completed postgraduate studies (one of them being a university professor), many others were self-taught and had received either no formal training in computer science or no university-level training at all. Many active members of SPIP studied in academic fields unrelated to programming, such as sociology, theology, classical literature, mathematics, industrial engineering, and theoretical computer science.

The projects also distinguish themselves by their values and political cultures. It's quite easy to see an activist tendency in SPIP, especially when compared with the inherently commercial orientation of the Symfony project. The SPIP charter, for instance, which outlines the principles for contributing to the project, explicitly states the following values: "to promote freedom of expression for all on the Internet," "a mistrust of money," and "the respect of everyone's identities" (translation by myself).[6] The charter also indicates that priority should be granted to community uses rather than commercial ones. Unlike SPIP, which is an independent project, Symfony is sponsored, as previously stated, by a private company, Sensio Lab. The governance of the project relies solely on the leadership of its creator, who is also the founder and executive director of the sponsoring company.

The organizational styles of annual public meetings also reflect these differences in terms of values. The annual Symfony conferences cost 200 euros, assemble hundreds of people, and are sponsored by several software companies, including Microsoft and Yahoo. The annual "SPIP parties," alternatively, are organized on a collaborative and collegial basis and are significantly less populated. The program of presentations is organized collaboratively just a few days before and demonstrates a flexibility that accommodates the contingencies of the day.

In some aspects, the differences between SPIP and Symfony also map the "free software versus open source" controversy that is very present in the free/open source community (Stallman, 2009). Indeed, while SPIP defines itself as a "logiciel libre" (free software), people in Symfony mostly talk in terms of open source. This free software versus open source controversy is also expressed in the licenses used in each project. The preferred license in SPIP is the GPL (GNU General Public License), which forbids

[5] In both cases, actors are mostly White Euro-American males. Women, though a significant minority in both projects, as is the case in many open source projects, demonstrate greater involvement in SPIP than in Symfony. For instance, I only saw two or three women at a Symfony conference attended by around 300 people. At a similar, but smaller SPIP event (50 people), a dozen of the attendees were women.

[6] http://zone.spip.org/trac/spip-zone/wiki/CharteDeFonctionnement

the reintegration of free source code in proprietary software. In contrast, the use of the GPL is prohibited in Symfony in favor of licenses like MIT or BSD that permit reusing open source code in the development of proprietary, closed-source applications.

These two projects were chosen for "opportunistic" reasons, since I took the opportunity of being in France to conduct my fieldwork there. I decided to conduct two case studies of software projects that were mainly written in PHP. Although I knew of some distinctions between these projects from the start, a more systematic comparative dimension of the analysis emerged later in my study, to produce contrasting effects on how the design of source code changes depending on the cultural and political perspectives of the actors involved.

Data were obtained through analysis and observation, for a period of one year, of the users and development mailing lists of each project as well as several online spaces and blogs where actors shared and discussed source code. I also conducted semistructured interviews with around 10 people in each project, selected for their diversity of roles and contributions—core developers of the projects but also members of the community who use the software to make websites (see Table 1). Finally, I attended events related to each project, which afforded me a better overview and contextual understanding.[7]

For my analysis, I follow the trend of inquiry first pursued by scholars of science and technology studies, such as Akrich (1992) and Woolgar (1991), that involves looking at how actors engaged in a design process "configure the users" by defining their tastes, preferences, and competencies, and elaborate a "script" that defines their roles and possible actions. These approaches are not intended to be deterministic. The script, for instance, is not thought of as absolutely binding. However, according to Akrich (1992), much of design work consists in "inscribing" a certain vision of the world into the technical content of the new object. The work of the analyst is therefore to reveal the negotiations between actors as they imagine the attributes of potential users, and to analyze how the determinations of these negotiations are expressed in a technical form. It is this approach that I follow in my analysis, by first looking at how these values appear in source code in a general way. I then continue by analyzing two debates, one in SPIP and another in Symfony, where the configuration of users is debated.

[7] I must also mention that I received training in computer science in my early career, which surely helped me better understand what was at stake in debates about coding choices.

Table 1. List of Interviews Done in the Study.

ID	Date	Duration of interview	Sex	Age	Role in the project/community
SPIP Interviews					
spip01	July 2009	178 min	m	39	Developer/member of the core team
spip02	August 2009	65 min	m	~40	User (who develops websites)
spip03	August 2009	96 min	m	~50	Developer/member of the core team
spip04	August 2009	64 min	m	19	User/creator of skeleton
spip05	February 2010	89 min	m	52	User
spip06	March 2010	135 min	m	~50	User
spip07	March 2010	111 min	m	60	User
spip08	April 2010	91 min	f	35	User/member of the core team (not as programmer)
spip09	April 2010	151 min	m	~30	Plug-in developer/server administrator
spip10	April 2010	92 min	m	28	Plug-in developer
spip11	May 2010	102 min	m	37	Developer/member of the core team
Symfony interviews					
sf01	June 2009	40 min	m	~30	Plug-in developer
sf02	June 2009	49 min	m	35	Developer/member of the core team
sf03	July 2009	81 min	m	35	Former member of the core team
sf04	July 2009	84 min	m	~35	Plug-in developer/user
sf05	March 2010	131 min	f	~25	Trainer
sf06	March 2010	97 min	m	24	Plug-in developer/patch contributor
sf07	April 2010	77 min	m	22	Trainer/plug-in developer
sf08	April 2010	87 min	m	~30	Former member of the core team/developer
sf09	April 2010	76 min	m	28	Plug-in developer
sf10	April 2010	61 min	m	44	Developer/community builder

Values in Code: A General Overview

Beyond identifying the values and political culture of each project, I want to focus on the way these organizational and political differences get *inscribed* into source code and its surrounding artifacts. One of the most obvious demonstrations of the inscription of values in source code relates to the choice of the natural language in which to write source code. Whereas all the code, comments, and documentation in Symfony are written in English, the writing of source code in SPIP looks more like a mixture of both French and English. Figures 1 and 2 show excerpts of source code from each project to emphasize this linguistic difference.

```
/**
* Constructor.
*
* @param array  $defaults    An array of field default values
* @param array  $options     An array of options
* @param string $CSRFSecret  A CSRF secret
*/
          public function __construct($defaults = array(), $options =
                         array(), $CSRFSecret = null)
                                       {
                        $this->setDefaults($defaults);
                          $this->options = $options;
```

Figure 1. Excerpt from Symfony source code.
Retrieved from http://trac.Symfony-project.org/browser/branches/1.4/lib/form/sfForm.class.php

```
// http://doc.spip.org/@exec_admin_vider_dist
function exec_admin_vider_dist()
{
        global $quota_cache, $spip_lang;

        // autorisation a affiner
        if (!autoriser('configurer', 'admin_vider')){
                include_spip('inc/minipres');
                echo minipres();
```

Figure 2. Excerpt from SPIP source code.
Retrieved from http://core.spip.org/projects/spip/repository/raw/branches/spip-2.0/ecrire/exec/admin_vider.php

The choice of "human" languages (between French and English) has political and social implications. The fact that SPIP is largely unknown outside of France, whereas Symfony is well known, is at least partly linked to their respective choice of language. In SPIP, many actors stated that they felt more comfortable writing code in French and thus felt that the community was more inclusive of non-English members. In Symfony, actors said that English was without a doubt *the* language of computing. The following quotes show two opposing views on the choice of language used in each project.

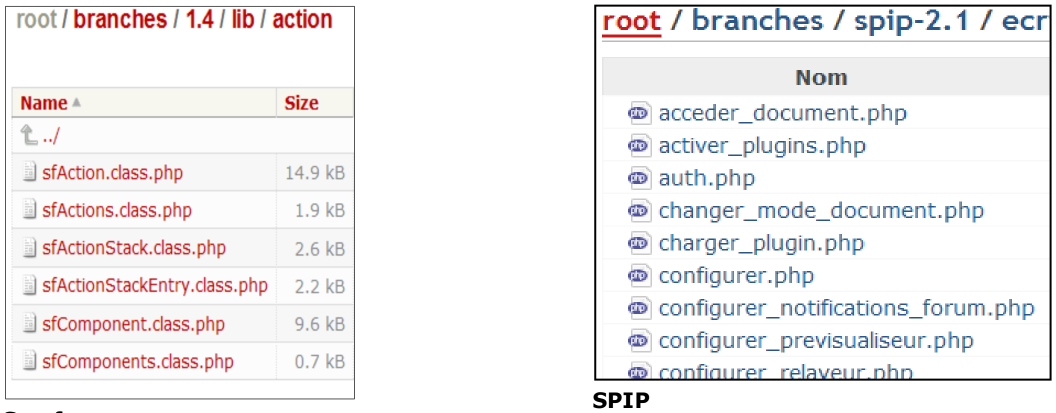
> I am convinced that we invent only in our own language, the imagination, the new ideas. Except if we are really bilingual. Forcing ourselves to use a language in which we are not completely at ease is a break to inventiveness. (Interview with spip03, August 2009)[8]

> For me, the language of computing is English. In software projects, everything should be in English. . . . All people who develop software should speak English. It's an absolute condition for being in the universe of computing. (Interview with sf06, March 2010)

More subtle differences also exist between the projects—for example, in their use of what programmers call "naming conventions," the standards that a project adopts for the nomenclature of the source code components. Naming conventions (in this case, for naming files) are much more normalized and strictly enforced in Symfony than in SPIP. Whereas in Symfony all file names should be in English, start with the *sf* prefix, and finish with class.php, in SPIP names should describe the file's function using a relevant French verb (*configurer*: to configure, *charger_plugin*: to charge plug-in). Figure 3 shows screen captures of the source code repository of each project and highlights the difference in naming files (the significance of these differences will be explained shortly).

The *sf* prefix used in the Symfony naming convention is also of particular interest. SF means Symfony but also harkens back to the original name of the project *Sensio Framework,* which, in turn, refers to Sensio Labs, the company that started and continues to sponsor the project. Other parts of the Symfony source code, most notably plug-ins, are named either after the individual who originally wrote the source code or the company that sponsored the specific work. For instance, the file name *lcopeninviterpostinstall.class.php* is prefixed by the two letters *lc* after LetsCod, the Lebanese Web agency that wrote it. In SPIP, however, the charter forbids the promotion of a commercial enterprise in the course of the project. Such a naming choice for a part of source code would thus be forbidden.

[8] All quotes from fieldwork have been translated from French to English by myself, unless otherwise stated.


root / branches / 1.4 / lib / action

Name ▲	Size
../	
sfAction.class.php	14.9 kB
sfActions.class.php	1.9 kB
sfActionStack.class.php	2.6 kB
sfActionStackEntry.class.php	2.2 kB
sfComponent.class.php	9.6 kB
sfComponents.class.php	0.7 kB

Symfony

root / branches / spip-2.1 / ecr

Nom
acceder_document.php
activer_plugins.php
auth.php
changer_mode_document.php
charger_plugin.php
configurer.php
configurer_notifications_forum.php
configurer_previsualiseur.php
configurer_relayeur.php

SPIP

Figure 3. File names in Symfony and SPIP.

Although more difficult to show with excerpts of source code, the organization of source code is very different in each project. On the one hand, Symfony portrays itself as a project that strongly emphasizes “clean code” and “best practices.” In particular, it puts a great deal of effort to reuse and integrate existing pieces of source code instead of “remaking the wheel.” Symfony development also strongly relies on *design patterns* of software development, a set of programming practices based on the book of the same name published in the 1990s and famous among computer programmers (Gamma, Helm, Johnson, & Vlissides, 1994). On the other hand, SPIP is often criticized for the messiness of its source code (one actor of SPIP qualified it as “not very beautiful”) and the fact that it avoids the more “modern” programming practices, such as “design patterns” or object-oriented programming. Indeed, SPIP seems isolated from all the discussions about standards and best practices. Indeed the SPIP community invented its own language—the SPIP skeleton language, more simply called the “SPIP language”—which is used to customize the appearance of a website made with SPIP. Interestingly, this language does not follow any recommended standard, something that is subject to debate within the community, as I will later show.

What could explain these differences of style in software development within each community? The main explanation may be the age of each project. SPIP was born at the turn of the millennium when Web programming was still in its infancy and best practices were not yet stabilized. Indeed, one actor of Symfony told me that these style differences reflect the difference between early users of PHP, who use this language because it is very easy to learn (he called them “Sunday coders”), and more recent “professional” PHP programmers, who use this language in the same way that they would use object-oriented languages like Java or .Net. Indeed, as I wrote previously, SPIP continues to attract contributors that come from different disciplinary horizons and are not necessarily familiar with best practices of professional programming.

However, some actors defend the design choices made in SPIP. In 2008, a blog post and subsequent comments criticizing SPIP received a strong reaction from one important SPIP contributor (who also happens to be a university professor in theoretical computer science). He commented that object-oriented programming and the so-called best practices of modern programming were in fact a fashion trend that never proved their inherent superiority over programming practices that "have less resorted to all sorts of intoxicating advertising" (his words, my translation) to promote themselves. More important for my argument, it is because of the values of collaboration and openness that SPIP does not succumb to the new programming practice fashions:

> SPIP is a collaborative project that aims to be open to as many people as possible; therefore, imposing to those who want to collaborate an encyclopedic knowledge of computer languages and practices goes against its objectives. . . . This is what has prevented SPIP to give in to fads, and allows it to attract as many contributors who do not claim to hold the truth on computers, and have fun together in advancing the project. (Comment on a blog post, December 2008,[9] my translation)

As the next section shows, the choices around these languages are debated within each community, and these debates put forward different figures of users.

Debating Values, Potential Users, and Design Choices: A Study of Two Debates

This section analyzes two debates—one in each community—which illustrate how values are mobilized into source code through both writing and meta-decisions that determine how the writing itself is performed. These debates surround the choices of format and language used to configure web interfaces, in SPIP and Symfony. It is important to note that these parts of the source code have the important common particularity of being primarily used by people outside the core development team of each project. In other words, the people who write or develop these parts of the source code are distinct from the people who use it. This is a classic distinction between users and designers, to which I will return later.

Methodologically, as previously mentioned, I follow the STS approach of analyzing how preferences and skills of potential users are negotiated and inscribed in a technological form. According to Akrich (1992), much of design work consists in "inscribing" a certain vision of the world into the technical content of the new object: "Designers thus define actors with specific tastes, competences, motives, aspirations, political prejudices, and the rest, and they assume that morality, technology, science, and economy will evolve in particular ways" (p. 208). Following this, I use the analytical approach described earlier by looking at the negotiations concerning the attributes of potential users and how the results of these negotiations are expressed in a technical form.

[9] In coherence with the rest of the paper, I avoid naming the blog to preserve the anonymity of participants.

SPIP: A Debate Over an Alternative Syntax for the SPIP Language

SPIP has developed its own programming language, called, alternatively, the "SPIP language" or the "SPIP skeleton language." This language is derived from HTML, but with some customized additions. It is used for configuring a website's front-end interface, with a bit more power than a "pure" template language by enabling simple database queries to display information in a dynamic way.

For many actors involved in SPIP, the skeleton language is one of the main added values of SPIP compared with other projects, since it enables those lacking deep programming skills to participate in the shaping and advanced configuration of their website. In a way, the SPIP language enables what Nardi (1993) calls "end-user programming," which allows end users to "create useful applications with no more than a few hours of instruction" (p. 5). Nardi notes that in end-user programming systems (that include, e.g., spreadsheet macros and statistical packages), a subset functionality can quickly be learned to get the job done. An SPIP contributor I interviewed describes this empowerment capacity of the SPIP language:

> With SPIP, what is extraordinary, is that if you don't code, you will succeed in doing stuff as if you were coding. All was done in a way that even if you don't understand anything you can manage it. Because you have a simplified language of loops that allows you to do what you could do with PHP, but in a simple and accessible way. So, it is accessible, and we are not in an elitist domain where we need to have done three years of hard coding to understand, to do something. (Interview with spip08, April 2010)

The debate that I analyze concerns a proposed change to the SPIP language syntax, designed to make it more consistent with globally recognized standards, most notably XML. Grammatically, the SPIP language, as it is now (or at least at the time of my study), is basically a derivation from HTML, with some custom instructions called *balises* in French (tags, in English). It is a "hacked" HTML in the sense that it is a set of custom and often ad hoc additions to HTML markup language, rather than a well-thought-out formal language. The controversy, which started during one of the yearly SPIP party conferences, relates to an attempt to standardize the SPIP language by adding a particular tag, <?spip, that would mark the distinctions between HTML and SPIP. Figure 4, copied from a document that was shared during this debate, shows an example of the current SPIP language syntax (above), and how the proposed alternative (below) would alter it. The most important proposed change to the current syntax is the <?spip code, which indicates a change of language, in this case from the SPIP language to HTML.

The person who proposed the change explained that an important problem with the current syntax is that both level of languages are indistinctly mixed. More pragmatically, this lack of distinction between different languages also makes it hard for some tools, like text editors, to work correctly. Clearly distinguishing SPIP language from HTML would allow the source code to conform to more globally recognized standards, most notably XML, and help the SPIP language be publicly recognized as a "real" programming language beyond its current user base.

Current syntax:

```
<B_art>
     <ul>
<BOUCLE_art(ARTICLES){id_article}>
     <li>#TITRE</li>
</BOUCLE_art>
     <ul>
```

Proposed alternative syntax:

```
<?spip AVANT art ?>
  <ul>
<?spip BOUCLE art ARTICLES { (id_article) ?>
  <li>#TITRE</li>
<?spip } art ?>
   <ul>
```

Figure 4. Current syntax and proposed alternative syntax.

Other members of the project strongly criticized the proposed new syntax, because of the "heaviness" of using the <?spip tag everywhere. Other people mentioned that the new syntax brought a perception of increased complexity to the language. Very interestingly for my argument, tropes like "nonprogrammers" and "novices" were mobilized to specify the figures of the actors that would be alienated by these changes:

> A language that is predictable to this point would likely be extremely rigid in its syntax, and thus discouraging to the *novice* and heavy to handle. (Spip-dev list, September 2009, emphasis added)
>
> If, once again, we have put ourselves in place of the *neophyte*, I think it's a subtlety that instead risks scaring him (Spip-dev list, September 2009, emphasis added).
>
> For a *new user*, it is a language that is difficult to digest, like coming into a world of complicated programming (such as <?php) (Spip-dev list, September 2009, emphasis added).

Although most of the interventions were along the line of defending nonprogrammers, other interventions argued the complete opposite, that is, to make the writing of "skeletons" more accessible to PHP developers:

> Would it make sense to also allow writing skeletons with a purely php syntax? Spip would then function simply as an API [application programming interface] and would be more easily accessible to "hard" *PHP developers* who would not have to learn a new syntax. (Spip-dev list, July 2009, emphasis added)

These comments show that choices regarding the internal design of source code involve what Woolgar (1991) characterizes as an act of "configuring the user." Recall that for Woolgar, technological design amounts to a process of debating and constructing a "figure of the user" that will eventually be embedded (or inscribed) in the machine (or the technical object). For Woolgar, "the capacity and boundedness of the machine take their sense and meaning from the capacity and boundedness of the user" (Woolgar, 1991, p. 68). In other words, the machine and its users are mutually defined, since designing the machine involves deciding how the users' actions will be facilitated or constrained. The study of the previous debate highlights this process of "mutual" definition of machine (source code, in our case) and user: The design choices of the SPIP language or the configuration formats are closely linked to how actors imagine and define their presumed or intended users, whether they are professionals, novices, or something in between.

This debate also shows the tensions between different sets of values when negotiating design choices. Some values relate to conformity with recognized standards and the desire to interact with the global programming community, whereas others prioritize inclusiveness and accessibility for people without a strong background in computer programming, even if it means being isolated from the "modern" programming community. For people participating in SPIP, it is possible—and important—to create a computer language, and by extension, source code, that may be accessed by novices and other nonprogrammers. This has important implications for this analysis, to which I will return later: It shows that source code is not—at least according to actors involved with SPIP—an artifact whose use is, in essence, limited to programmers alone. Instead, the agency of source code users is linked to the design of the accessibility of software source code with which they interact.

Following this short, but intense debate, the community did not follow up on the proposal and decided to stay with the current SPIP language.

Symfony: Controversy Over the "Forms" Framework

The debate that I analyze in Symfony turned into a full-blown controversy. In 2009, just before my research began, one of the actors heavily involved in the project decided to leave after disagreeing publicly about certain changes in the software source code. This departure created a commotion in the community. During an annual conference, several participants raised concerns about the impact of this departure on the community and its software project. A year after his resignation, the actor decided to participate again in Symfony, only this time in a more peripheral role. Contrary to the SPIP controversy presented earlier, in which the proposed changes were finally rejected, in the case of Symfony, the proposed changes were carried on despite the controversy.

The controversy concerned a new way of configuring and managing Web forms introduced in version 1.1 of Symfony. "Forms" refers to any input field in an interactive website, like login or comment boxes. In Symfony 1.0, forms were primarily managed using a programming language called YAML (YAML is not markup language) that was considered simple to read and use. In a subsequent version, this way of making forms was abandoned in favor of an approach that required much greater skills in programming. Figure 5 is excerpted from a blog post criticizing the change and shows the two ways of implementing a form in Symfony.

Configuration in PHP format (new 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(),
```

Configuration in YAML format (old 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 }
```

Figure 5. Different formats for configuring the forms in Symfony.
Excerpted from a blog post commenting the change.

This controversy is relevant to analyze because, as with the previous case, it concerns some parts of the source code that are primarily used by people outside of the core development team. It also confronts different visions of the development of the software project. The following interview quote explains the different positions in the controversy:

> The confrontation that B had with G was about, uh, the ergonomics of the code . . . . That is to say, he wanted to be able to configure forms and procedures in a language which is called YAML . . . which is a bit like XML, but in a much more readable syntax. And the arguments of G toward the proposal has been to say, "I still prefer that people write PHP," that is to say, the language operated with the framework to describe complex things. (Interview with sf08, April 2010)

The controversy thus concerns the *ergonomy* of the code, most specifically whether the code should remain accessible and "readable" or should be changed to better do complex things. The following is a quote from an interview with an actor who defends the new form system. He considered that the added values of security and scalability justify the increased complexity:

> It is complicated, but it has the great advantage of structuring a lot of things, and to ensure safety and scalability. And I am convinced that this is a good thing, and certainly plenty of people think it is a bad thing because it became too complicated. (Interview with sf02, June 2009)

In contrast, one actor, in favor of maintaining the YAML format, argued that the changes have harmed users:

> It seemed to me that Symfony initially responded to this need for simplicity for developers. . . . For the following versions, we wanted to take a little more height and restructure it in a slightly different way. For that, we made some changes that I think . . . interfered with the ease of use of certain points. (Interview with sf03, July 2009)

For this actor, taking "a little more height" was associated with the quest to make "beautiful" code. However, he saw this goal as potentially harmful when actually using this code to make a website:

> I put myself in the place of the developer, and I think, ah, it's beautiful code . . . when you see it. But when you start to use it you find that, finally, it is not very practical! They could have made it a little less beautiful, but easier to use! (Interview with sf03, July 2009)

One actor who defends the change of format acknowledged that the increased complexity of the framework is one of the main sources of contention. However, he justified this complexity by suggesting that the framework is not intended for everybody:

> So, the tensions, we have them because there are people who find that Symfony is much too complicated. And what I would like to say is that, it's okay. It simply means that the framework is not made for these people. (Interview with sf02, June 2009)

This last quote shows how the complexity of the framework might be assumed to exclude certain types of users. Most specifically, one actor (sf02) told me that Symfony's target users are not made up of hackers (*bidouilleur*, in French)—"those people who do not necessarily make a living with programming, but use PHP because it is very simple to learn"—but rather of "professionals," "those people who use PHP in commercial companies" (Interview with sf02, June 2009). This interview quote frames the issue clearly:

> The target users of Symfony, Zend framework or EasyComponents, is a very limited target compared to the global community of PHP. This is a rather professional target, people who do this in their job, so, people who have a quite important technical background. (Interview with sf02, June 2009)

As was the case in SPIP, this controversy shows how different visions of users conflict within the project, and how one vision eventually gets configured within source code, whereas the old architecture was seen as easier to use for the "integrator" or the developer—seen as a general figure—the change of architecture is now directed to the figure of the professional. These figures of users are also intertwined with a specific design of source code as well as with particular values such as beauty, simplicity, security, and scalability.

Discussion: "My Interface Is Code," or Designing Source Code for Potential Users

The analysis of the two controversies shows parallel behavior of enforcing the project's policies and norms in the design of source code. It also shows how the design of source code distinctively reflects each project's values and political culture. On the one hand, the SPIP community decided to keep the SPIP language as it was, that is, as a sort of hacked language that does not follow best practices or conform to global standards of programming languages, but remains accessible (at least according to actors) to people with a broad spectrum of skills and backgrounds. The SPIP code remains, in a sense, faithful to its original activist and associative values of inclusion, collaboration, and accessibility. On the other hand, from the perspective of some actors, Symfony broke with its perceived former aim at simplicity to reach more professional and skilled users and to conform to best practices in Web programming. Both debates also show how the agency of certain categories of actors (e.g., those without an important programming background) may be affected by design choices that are integral to the development of source code itself.

Looking at the design of source code, with concepts such as scripts and configuration of users, also brings us to consider forms of interaction with source code that are closer to usage than simply designing. Indeed, there is a tendency in communication studies and other fields to see software programming exclusively as a design activity. The users are generally the ones using Facebook, YouTube, and Twitter to produce "content," whereas the "code writers" are the ones designing tools or making the Internet's "laws," to borrow Lessig's (2006) words. However, as stated by Myers et al. (2016) in the title of their article "Programmers Are Users Too," when one builds a website, for instance, one does not build everything from scratch, but rather uses some existing source code to accomplish this task. This notion of "users of source code" is even more striking in the cases I presented earlier because the actors involved in the debates do not all have the same relationship to particular pieces of source code: Some actors are designers, whereas others are users. As I previously mentioned, the pieces of source code debated in the controversies are specifically aimed at configuring software that enables others to develop a website. Those who write these pieces of source code will not necessarily use them themselves to develop a website; others will use them to develop their website, but not necessarily contribute to their development (although some of these users participate in discussions related to coding choices, as was the case in the debates I analyzed).

Indeed, the users of source code to whom I am referring are very similar to Akrich's users in the case study that first brought forward the *script* concept (Akrich, 1992). Recall that in this case study, Akrich looked at how a photovoltaic lighting kit used in Africa was designed to give little freedom to users to repair or transform it. Akrich considered the technicians who were able to repair the lighting kit as a figure of users who were excluded from the design of the artifact in favor of docile users who would not make any changes to the lighting kit. Akrich's analysis is thus interested not so much in end users, but rather in users who—although not the primary developers of the product—are able to interact with deeper technical aspects of the object to reconfigure or assemble it in a larger sociotechnical network.

This distinction between users and designers also brings the focus to the agency of particular pieces of source code that act as interfaces between these categories of actors. I come back to my introductory quote by a study participant who explicitly compared source code with an interface, with which he interacts to modify software: "When I am a graphic designer, my interface is Photoshop, with its buttons, its windows, etc. When I am developer, my interface is the code. It is through code that I interact with what I am building, a program" (Interview with sf03, July 2009). For this actor, source code may—and should—be evaluated using the same categories as those used to analyze other user interfaces: usability, ergonomy, accessibility, and so forth. Again, this echoes the statement made by Myers et al. (2016), which I quoted before, to the effect that "software development languages and environments are the user interfaces through which developers interact with computers" (p. 1). This also echoes Galloway's suggestion, within media studies, that source code is an interface, understood as a general means of mediation between subjects and objects. Source code, Galloway (2012) writes, "is a kind of interface, an interface into a lower level set of libraries and operation code" (p. 9). In this way, a particular piece of source code acts as an interface between other parts or "layers" of the larger set of source code but also as the material through which it is possible to modify a software program or digital object. Like other interfaces, it is important to look at how the design of source code may reflect values or participate in the enabling or restraining of participation in the making or reconfiguration of digital media and technologies.

Conclusion

In an article addressing the relationships between formal languages and social relationships, Vaucelle and Hudrisier (2010) note that the future of the Web will not rely that much on technical improvements made by computer scientists, but instead on the social appropriation by individuals who are today children and teenagers and who will handle highly mutant and convergent interactive media objects. In this perspective, they write, computer languages and structured programming play a prominent role as they facilitate the integration of different forms of interaction that are still very heterogeneous today. For these authors, the future of interactivity depends not only on technological developments but also on linguistic mutations that remain, at this time, largely unpredictable. The increasing importance being accorded to fluency in programming languages is also seen in the rise of popular education on this subject. For instance, in 2012, *The New York Times* published the "Surge in Learning the Language of the Internet," about the booming market for classes and online courses in Web programming and mobile applications (Wortham, 2012). These skills, the author wrote, are increasingly in demand as the need to customize a blog or fill a database itself increases.

As technologies such as interactive websites and mobile applications are increasingly relational and configurable, more and more people are looking to learn programming skills that will enable them to individually or collectively improve their digital environment. Moreover, as academic programs in communication studies include courses or even specializations in interactive media, programming skills are becoming more relevant, not only to professional computer programmers but also to professional communicators who often have to build and adapt websites and digital technologies. It is in this context that it seems important to me that researchers—especially in communication studies—engage in analyses that look at how people use source code, and how its design may potentially exclude or facilitate the participation of some actors or how it may embody specific values.

Although some empirical work has been done in this perspective in such fields as HCI, these studies have been mostly conducted within computer science research groups. So far, reflection on source code within communication studies seems to have stayed on the theoretical or discourse level, without seriously considering what it means to use and interact with source code. The contribution of this article is to provide an example of how to address these issues, from an empirical perspective. Of course, more work needs to be done to understand the impact of source code design choices on the agency or empowerment of actors, but the analysis put forward in this article shows that these are relevant issues for actors involved in these projects and that source code should be critically assessed in the same way as other interfaces.

References

Akrich, M. (1992). The de-scription of technical objects. In W. E. Bijker & J. Law (Eds.), *Shaping technology/building society* (pp. 205–224). Cambridge, MA: The MIT Press.

Chun, W. H. K. (2008). On “sourcery,” or code as fetish. *Configurations*, *16*(3), 299–324.

Coleman, G. (2009). CODE IS SPEECH: Legal tinkering, expertise, and protest among free and open source software developers. *Cultural Anthropology*, *24*(3), 420–454.

Coleman, G. (2012). *Coding freedom: The ethics and aesthetics of hacking*. Princeton, NJ: Princeton University Press.

Demazière, D., Horn, F., & Zune, M. (2007). The functioning of a free software community: Entanglement of three regulation modes—Control, autonomous and distributed. *Science Studies*, *20*(2), 34–54.

Free Software Foundation. (2007, June 27). GNU general public license. *GNU Operating System*. Retrieved from https://www.gnu.org/licenses/gpl-3.0.en.html

Friedman, B., Kahn, P. H., & Borning, A. (2008). Value sensitive design and information systems. In K. E. H. Herman & H. T. Tavani (Eds.), *The handbook of information and computer ethics* (pp. 69–101). Hoboken, NJ: Wiley-Interscience.

Fuller, M. (2008). *Software studies: A lexicon*. Cambridge, MA: The MIT Press.

Galloway, A. R. (2006). Language wants to be overlooked: On software and ideology. *Journal of Visual Culture*, *5*(3), 315–331.

Galloway, A. R. (2012). *The interface effect*. Cambridge, UK: Polity.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software*. Reading, MA: Addison-Wesley Professional.

Kelty, C. (2008). *Two bits: The cultural significance of free software*. Durham, NC: Duke University Press.

Knobel, C., & Bowker, G. C. (2011). Values in design. *Communications of the ACM*, *54*(7), 26–28.

Krysia, J., & Grzesiek, S. (2008). Source code. In M. Fuller (Ed.), *Software studies: A lexicon* (pp. 236–243). Cambridge, MA: The MIT Press.

Lessig, L. (2000). *Code and other laws of cyberspace*. New York, NY: Basic Books.

Libre-Fan. (2008, June 20). SPIP, dépassé? [SPIP, outdated?]. *Libre-Fan blog*. Retrieved from http://librefan.eu.org/node/138

Marino, M. C. (2010, September 15). Critical code studies and the electronic book review: An introduction. *Electronic Book Review*. Retrieved from http://www.electronicbookreview.com/thread/firstperson/ningislanded

Montfort, N., Baudoin, P., Bell, J., Bogost, I., Douglass, J., Marino, M. C., & Vawter, N. (2012). *10 PRINT CHR$(205.5+RND(1)): GOTO 10*. Cambridge, MA: The MIT Press.

Myers, B. A., Ko, A. J., LaToza, T. D., & Yoon, Y. (2016). Programmers are users too: Human centered methods to improve software development. *IEEE Computer, Special Issue on UI Design*, *29*(3), 44–52.

Nardi, B. A. (1993). *A small matter of programming: Perspectives on end user computing*. Cambridge, MA: The MIT Press.

Nissembaum, H. (2005). Values in technical design. In C. Mitcham (Ed.), *Encyclopedia of science, technology and ethics* (pp. lxvi–lxx). New York, NY: Macmillan.

Shilton, K. (2013). Values levers: Building ethics into design. *Science, Technology & Human Values*, *38*(3), 374–397.

Souza de, C., Froehlich, J., & Dourish, P. (2005). Seeking the source: Software source code as a social and technical artifact. In *Proceedings of the 2005 International ACM SIGGROUP Conference on Supporting Group Work* (pp. 197–206). New York, NY: Association for Computing Machinery. doi:10.1145/1099203.1099239

SPIP. (n.d.). *What is SPIP?* Retrieved from http://www.spip.net/en_rubrique25.html

Stallman, R. (2009). Viewpoint: Why “open source” misses the point of free software. *Communications of the ACM*, *52*(6), 31–33.

Suchman, L. (2007). *Human-machine reconfigurations: Plans and situated Actions*. New York, NY: Cambridge University Press.

Symfony. (n.d.). *Symfony. Open-source PHP Web framework*. Retrieved from http://www.symfony-project.org/index.php

Vaucelle, A., & Hudrisier, H. (2010). Langages structurés et lien social [Structured language and social link]. *tic & société, 4*(1). Retrieved from https://ticetsociete.revues.org/790

Woolgar, S. (1991). Configuring the user: The case of usability trials. In J. Law (Ed.), *A sociology of monsters: Essays on power, technology, and domination* (pp. 57–99). London, UK: Routledge.

Wortham, J. (2012, March 27). A surge in learning the language of the Internet. *The New York Times*. Retrieved from http://www.nytimes.com/2012/03/28/technology/for-an-edge-on-the-internet-computer-code-gains-a-following.html