Seeking the Source: Software Source Code as a Social and Technical Artifact

Cleidson de Souza[1,3]
[1]Departamento de Informática
Universidade Federal do Pará
Belém, PA, Brazil - 66075
cdesouza@ics.uci.edu

Jon Froehlich[2]
[2]Computer Science & Engineering
University of Washington
Seattle, WA, USA – 98195
jfroehli@cs.washington.edu

Paul Dourish[3]
[3]Donald Bren School of Information and Computer Sciences
University of California, Irvine
Irvine, CA, USA – 92667
jpd@ics.uci.edu

ABSTRACT
In distributed software development, two sorts of dependencies can arise. The structure of the software system itself can create dependencies between software elements, while the structure of the development process can create dependencies between software developers. Each of these both shapes and reflects the development process. Our research concerns the extent to which, by looking uniformly at artifacts and activities, we can uncover the structures of software projects, and the ways in which development processes are inscribed into software artifacts. We show how a range of organizational processes and arrangements can be uncovered in software repositories, with implications for collaborative work in large distributed groups such as open source communities.

Categories and Subject Descriptors
D.2.2 [**Software Engineering**]: Design Tools and Techniques – user interfaces. D.2.10 [**Software Engineering**]: Design – representation. H.5.3 [**Information Systems and Presentation**]: Group and Organizational Interfaces – computer-supported cooperative work.

General Terms
Design, Human Factors, Measurement.

Keywords
Social networks, software repositories, data mining, socio-technical systems.

1. INTRODUCTION
Studies of the social organization of technical work have repeatedly drawn attention to the complex interactions between social practice and technological artifacts [e.g. 2, 24, 31]. The artifacts that mediate and support technical work reflect not just technical constraints and needs but simultaneously reflect working arrangements, divisions of labor, and aspects of expected practice.

This dual role is exhibited not simply by technological hardware, but by other "technical" artifacts such as classification schemes and formal representations.

For example, Bowker and Star [4] discuss the social embedding of classification schemes. Drawing on a number of examples, but most particularly the International Classification of Diseases, they illustrate how classification schemes reflect a social order as much as a natural order, making the phenomena they describe amenable to forms of analysis, interpretation and computation that reflect the social arrangements of the work. Lynch [25, 26] explores the central role of image-making in scientific practice and communication, and discusses the "rendering practices" by which features of messy reality are transformed into more portable and broadly consumable visual forms, designed again for particular sorts of comparison and discussion. Fujimura [17] describes how scientific discoveries are transformed into standardized packages of techniques and technologies that allow them to be moved between sites, and which in turn influence how scientists see problems as "do-able" and amenable to particular solution approaches.

Latour refers to this phenomenon as "inscription"[23]. Drawing on anthropological studies of scientific laboratory practice [24], he describes how social arrangements, debates, divisions of labor, and patterns of work become inscribed into the artifacts and representations in which science trucks. Inscription is a process through which social practice and technological artifacts become inextricably intertwined. For example, standardized processes imply divisions of labor, standardizations of skill, etc. [34]; formal models imply ways of uniformly disambiguating between "interesting" and "uninteresting" (or "relevant" and "irrelevant") phenomena [3]; instruments and devices imply particular ways of working and available infrastructures [33]; and methods and models create conventional and acceptable ways of formulating problems [17, 18].

One domain of technological practice that has been of particular interest to researchers in CSCW is software systems development [13, 20, 21]. Inscription issues are particularly relevant to software development practice, since software artifacts are pure inscriptions; free from traditional physical constraints, they are written forms that describe the forms and patterns of software system structure and operation. Software mechanisms are, in general, subject to much less external constraint than physical mechanisms, which is a source of tension in joint hardware/software development teams [32]. In short, there are very many different ways of producing working software systems.

The discipline of Software Engineering is, arguably, primarily concerned with developing software systems that satisfy not simply internal functional constraints but also external constraints of modularity, reusability, maintenance, comprehensibility, documentation, etc, which themselves reflect organizational and social expectations of how, where, when, and why the software system may be used. For instance, Conway [9] recognized over 30 years ago that the structure of the system mirrors the structure of the organization that designed it, while Parnas defined a software module as "a responsibility assignment rather than a subprogram" [29].

Software development, then, is a particularly fruitful domain in which to study the relationship between technological artifacts and the social structures that shape them.

It is this relationship that drives the work presented here. In particular, we want to explore aspects of the relationship between software artifacts and software development processes. Although processes are more or less well-defined in formal organizations, informal software development, such as that associated with the free and open source software movement, has a different character. Open source projects must essentially produce their own structures. These structures are emergent rather than formal, implicit in the development practice rather than explicitly codified. While this allows open source projects to be flexible, it also makes them more complicated for participants to understand or explain; software development processes are a means for organizational accountability as well as organizational regulation [19]. Our work has been motivated by the question of whether aspects of informal software process can be found in the structure of the software artifact itself. Using a software visualization tool, Augur, we have been conducting an analysis of the artifacts of a number of software projects, a "software archeology" to explore the relationships between artifacts and activities as they are negotiated in distributed software development through mining software repositories.

The paper is structured as follows: in the following section we discuss in greater detail the sources of complexity in software development. Next, we present our visualization-based approach for analyzing software projects. We then explore open-source software projects in greater detail with examples. Finally, we discuss the implications of our results for the understanding of open-source projects, followed by some concluding remarks.

2. ARTIFACTS AND ACTIVITIES IN SOFTWARE DEVELOPMENT PRACTICE

Software development teams face two sources of complexity in their work – the complexity of the artifact, and the complexity of the activities that surround it. By the complexity of the artifact, we mean the inherent complexity of the software system – the appropriate design and use of algorithms, architecture, structure, parallelism, scale, dynamic behavior, etc. The creation of software is a skilled practice, and much of this skill is in understanding the opportunities and limitations of different approaches to technical problem-solving and different software designs. This sort of complexity is inherent in software system design; it characterizes even the creation of small-scale software systems by individuals, such as in programming assignments to be solved by students, personal programming projects, etc, and many empirical studies of programming practice have pointed to aspects of the problem-solving process [e.g., 11]. However, since most professional software development (and even much amateur software development, in the open source world) is conducted not by individuals but by teams, then a second source of complexity arises – the complexity of the development activities themselves. By this we mean the complexity introduced by the fact that multiple people are creating and modifying the software system at the same time, requiring developers to coordinate parallel and distributed work, identify, avoid and recover from conflicts, anticipate problems, share goals, formulate strategies, and achieve a coherent concerted effect. Many empirical studies of software development teams focus on these coordination problems [e.g. 22].

Many researchers have studied the complexity of software development practices with an eye to developing new technologies that can help developers deal with these complexities. Reflecting these two sources of complexity, two technological strategies have emerged to deal with them and help developers in their day to day work The first focuses on the complexity of software artifacts, and attempts to give software developers better tools for understanding, interpreting and manipulating the software artifact itself. For example, many forms of software analysis help the programmer to understand the structure of the software system. Software systems exhibit two forms of structure, static and dynamic. The static structure of the software system concerns the relationship between the units that the programmer creates – classes, methods, modules, variables, and other components out of which software systems are built. The dynamic structure of a software system concerns how this program will give rise to a running process – where the program's static structure specifies potential action, the program's dynamic structure concerns actual behavior. Each of these may be analyzed and made available to software developers as ways to help them in the process of software development. For instance, static analysis can uncover certain kinds of potential security concerns [35] and uncover potential error states that might arise at run-time, as well as helping programmers understand the relationship between different elements in the software system [5]. Dynamic analysis can be used to create profiles of a program's run-time behavior, determining which elements of the program consume the majority of memory, activity, etc. Especially when programs grow large, providing automatic tools to understand these structural properties of software systems can relieve developers of a considerable practical burden, and can help them to create systems that operate effectively.

The second focuses on the complexity of development activities. These tools are, perhaps, more familiar to researchers working in Computer-Supported Cooperative Work, since the mechanisms that have been developed to support software development have also been applied in other domains. The most widespread set of tools are those based on formal descriptions of the software development process – models that specify how the software development task is broken down into a series of sub-tasks, and how those subtasks are to be coordinated. Configuration Management (CM) systems describe how software systems are arranged, and the relationship between elements within a development process; they enforce rules that prevent simple conflicts from taking place by regulating access to the software artifact under construction (ensuring that only one developer can be working on a specific module at a time, for example.) Like workflow systems, these process-based software development systems impose an order on the software development activities in order to prevent breakdown situations from arising. An alternative approach has been to support more open-ended forms of coordination based on mutual visibility, awareness, and end-user coordination rather than formal process-based coordination. Of course, these two modes of operation are not exclusive. Grinter has particularly drawn attention to this issue; in her empirical studies of software engineering, she has noted how users of CM systems use the information it provides to maintain an informal awareness of each other's activity and to interpret and anticipate potential consequences for their own [19].

However, although these two sources of complexity (the artifacts and the development process) are typically addressed in isolation, empirical studies of software development practice suggest that, for programmers and developers, they manifest themselves as part of a common problem. For example, one of our recent studies looked at a team of software developers engaged in the maintenance of a software system called MVP [12]. This team used a state-of-the-art CM tool to manage their coordination issues surrounding the changes in the source code. However, in addition, they had to adopt an email convention that advised developers to send an email to the team's mailing list with a brief description of the impact that their work (changes) would have on other's work. By doing that, MVP developers allowed their colleagues to prepare for and reflect about the effect of their changes. This suggests that the software artifact being developed and its development activities need to be somehow integrated. In this case, the source-code dependencies affected the software development activities adopted by the MVP team. Aiming to address this problem by providing developers with a more comprehensive view of the software development process we developed Augur, that brings together views of the artifact and views of its surrounding activity. Augur is explored in the next section.

3. VISUALIZING SOFTWARE DEVELOPMENT

The explorations that we describe here have been conducted using a tool called Augur, a system for visualizing software systems [16]. Augur is a visualization system based on the Seesoft paradigm [15], in which properties of the software system are mapped to color and other features of a graphical display of the source code itself. For instance, in the most common case, we might show an overview of the system in which each line of source code is represented by an equivalent line of pixels, colored to indicate how recently each line was modified. This view allows a developer or manager to see which areas of the system are "active."

Our initial system, described elsewhere [16], integrated simple code analysis with analysis of activity records, and made these accessible in a single visual frame providing coordinated views. In addition to displaying the pattern of activity over the source code, it also displays aspects of the structure of the source code. This coordinated view allows developers to understand the character of the activity carried out – not just that a modification has been carried out, but what sort of a modification it is (the addition of a new method, code "commented out", a revision to existing functionality, etc.)

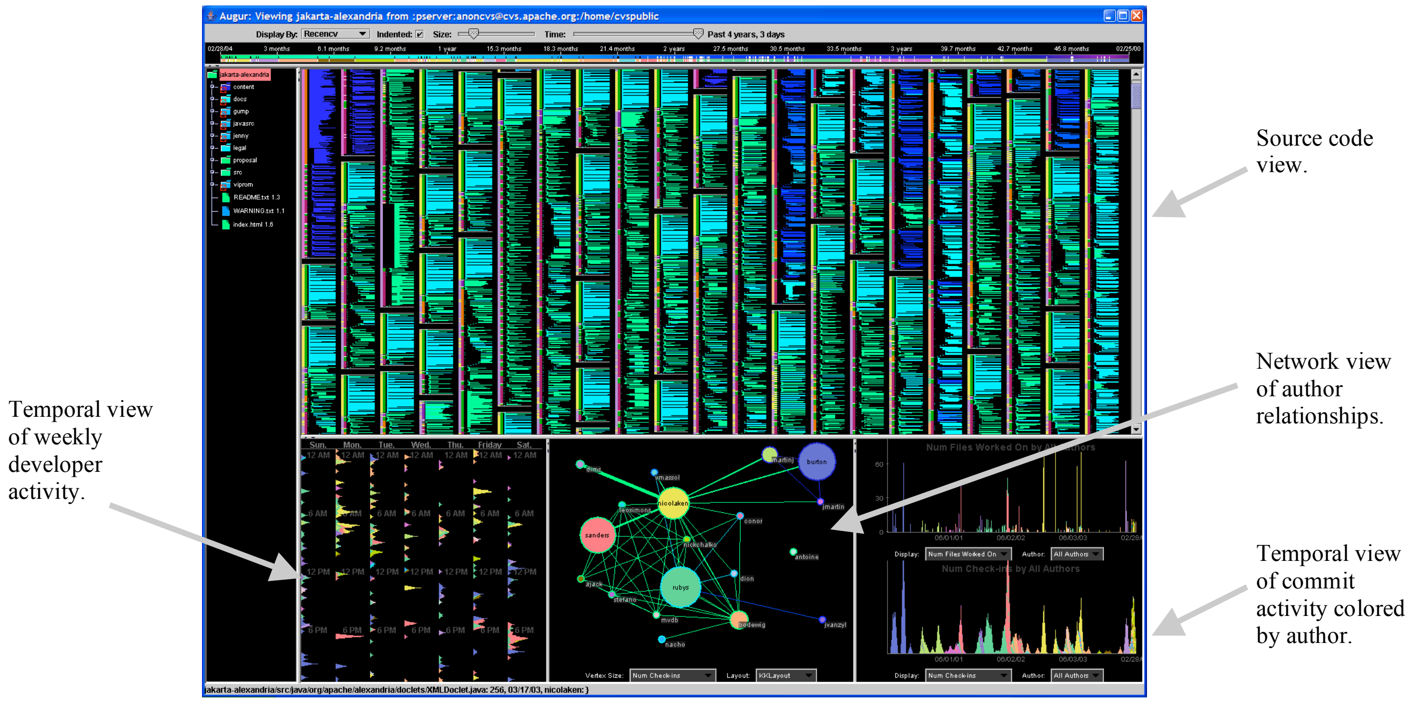

Augur: Viewing jakarta-alexandria from :pserver:anoncvs@cvs.apache.org:/home/cvspublic
Display By: Recency
Indented:
Size:
Time:
Past 4 years, 3 days
Source code view.
Network view of author relationships.
Temporal view of weekly developer activity.
Temporal view of commit activity colored by author.
Sun. Mon. Tue. Wed. Thu. Friday Sat.
12 AM
6 AM
12 PM
6 PM
Num Files Worked On by All Authors
Num Check-ins by All Authors
Vertex Size: Num Check-ins
Layout: KKLayout
Display: Num Files Worked On
Author: All Authors
Display: Num Check-ins
nicolaken
sanders
rubys
burton
jakarta-alexandria/src/java/org/apache/alexandria/doclets/XMLDoclet.java: 256, 03/17/03, nicolaken: }

Figure 1 – Augur

The basic Augur interface is shown in Figure 1. Each pane displays a different aspect of the system being examined: changes in one view are immediately reflected in the others. The large central pane shows the line-oriented view of the source code. In the figure, the color of each pixel line indicates how recently it was modified; this allows a developer, at a glance, to see how much activity has taken place recently and where that activity has been located.

In our informal evaluations, developers involved in distributed software development projects relied upon both the activity information and the structure information in coordination to develop a holistic view of software development activity.

As noted above, Augur was originally developed as a tool for software developers, providing them with a flexible visualization of system activity. However, as we explored the range of ways in which people used the tool, and the set of extensions that they proposed, we found that it might provide value too as an analytic tool. In particular, as users asked for mechanisms that would help them explore the structure of the system with finer granularity, we noted that these structures might also provide the basis for an "archeological" exploration of a software repository. In our more recent work, we have been extending the richness of the representations that Augur provides, in response both to user requests and to the new opportunities for analysis. In particular, we have been exploring the use of call-graph analysis and network analysis as ways of forming richer pictures of distributed development activities. Insights from our previous fieldwork with commercial software development [13] also suggest this avenue of research.

3.1 Call-Graph Analysis

The original version of Augur incorporated only a simple static form of structural analysis, one that classified lines of code according to their type, and so allowed a developer to see each line of code in terms of the larger structures within which it was embedded. In our more recent versions of the system, we have begun to augment this view with information that explores the dynamic structure of code.

In particular, we have incorporated call-graph analysis. A call graph is a data structure that describes which elements of a software system make use of which other elements. Software systems are constructed in terms of procedures (or "functions" or "methods"), which may in turn make use of the results of other procedures, just as, in mathematics, a function can be defined which makes use of the results of other functions (e.g. if $f(x) = sqrt(x) + 1$, then the function f makes use of the function *sqrt*). A call graph lists all the procedures in a software system, and, for each procedure, shows what other procedures it makes use of.

A call graph, then, reveals the potential dynamic structure of a software system, although it can be derived using static analysis techniques (i.e., it can be extracted directly from the source code, without examining a running instance.) More importantly, in demarking dependencies within the code (between one procedure and another), it also begins to suggest dependencies within the development team (between the maintainer of one procedure and the maintainer of another) [13].

3.2 Network Analysis

The relationship between members of the development team is made more explicit in the network view. In this view, Augur draws views of the network of contributors to a project, relating them according to patterns in their development activity. For example, a simple graph shows the relationship between project members who have contributed code to the same modules. This view abandons the source code as the primary spatial framework for displaying activity information; instead, it adopts a conventional graph (node and line) structure to show the relationships between people directly.

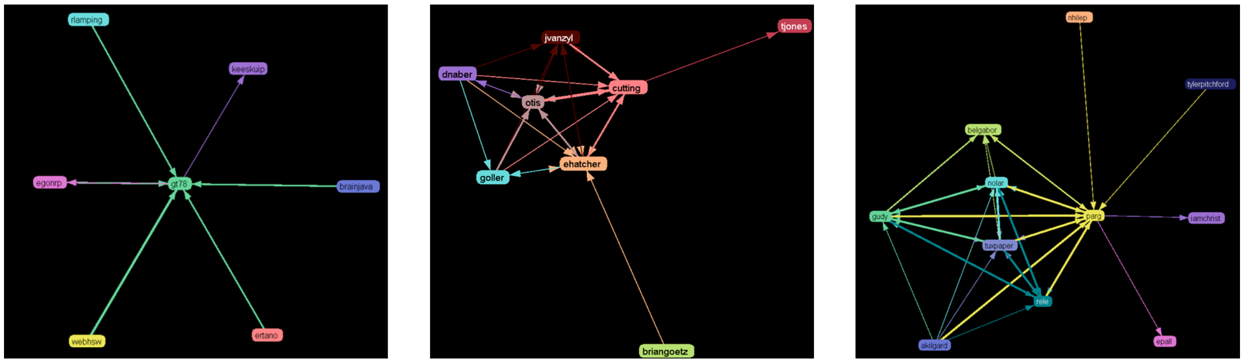
The network view can use different graphical properties to indicate different features of the relationship between individuals. For example, in the graphs shown in Figure 1, each node (circle) represents a specific individual, while the lines between the circles indicate that the developers have both contributed code to the same module. The size of the circle indicates how many lines of code someone has contributed, while the thickness of the line indicates how many lines they have contributed to files in common. Finally, the color of the lines indicates how recent this activity is, with brighter colors indicating more recent activity.

3.3 Combining Networks and Software Structure

While useful individually and in combination with the other views that Augur provided, we found that these two perspectives could be fruitfully combined to tackle the problem noted in our earlier empirical work – that is, the ways in which software developers must orient towards dependencies between their own work and the work of others.

Essentially, these perspectives highlight two sets of relationships in the software development process – the relationships between elements of the system (in particular, dependency relationships between different components), and relationships between the people who work on those components. Bringing these together begins to uncover the ways in which dependencies between parts of the software system can reflect or lead to dependencies between the developers themselves. They provide a technical means to explore the question that we raise at the start of this paper – that is, the extent to which software artifacts have inscribed into them patterns of interaction and participation. Augur, with these facilities, allows these questions to be explored empirically.

There are two ways in which we have been exploring this. First, we have combined the two sources of information to replace the module-dependency graphs that arise from the call-graph analysis with author-dependency graphs that detail the relationships between authors. In particular, this allows us to resolve some problems that the engineers in our field study need to resolve, which is to determine who is likely to be affected by upcoming changes (or, conversely, whose work is likely to have an impact upon my own.) Second, by using the revision history features of the underlying CM system, we are able to look at patterns in the evolution of both technical and social structure of the system – how people join and leave a project, how participation patterns change over time, and how these changes might be related to the evolution of the software system itself.



(a) "Centralized" (b) "Densely Networked" (c) "Core and Periphery"

Figure 2: Forms of participation

4. EXPLORING SOFTWARE PROJECTS

The particular focus for our analysis is open source software development. Open source is an approach to software system development in which loosely-knit collections of volunteers, collaborating over the public Internet, create software systems whose source code is available to all (rather than being protected as a trade secret, as it is in most commercial development.) Proponents of open source development models claim many advantages for this approach, both practically and politically, including faster and more responsive development cycles, and more secure and robust software products. Unencumbered access to the source code and the development process is the central feature of the open source model, although its details vary from project to project. Although it is most commonly associated with non-commercial software development, many open source projects are at least partially funded by commercial activities and involve professional software developers whose participation is sponsored by their employers [36]. However, since open source projects exist either partly or entirely outside an organizational context, development processes and procedures do not follow organizationally mandated models [1]. Indeed, in advocating open source development models, open source analysts explicitly contrast regularized management and oversight (what is called the "cathedral" approach) with the more informal and ad hoc arrangements of open source projects (the "bazaar"). Since open source projects must, therefore, evolve their own working arrangements and means of enforcing them, they are a particularly interesting object of study.

This section describes different analysis that we performed in different open-source projects using Augur, in particular combining network and software structure, as discussed in the previous section.

4.1 Types of Projects

By looking at the networks of relationships between developers as indicated by dependencies between code modules, we can see how different approaches to project organization are reflected or "inscribed" in the source-code itself of each project. In a centralized approach, the control is potentially reflected in a call-graph structure where other developers' code is called by the "architect"s code. This developer's code is the "glue" that connects the whole project together, all other developers' code does not interact among themselves. That is not to say that the other code is not important nor relevant, we are just arguing that the architect's code is the one integrating the whole project. Figure 2a illustrates this example; it is possible to identify a high degree of centralization around developer "gt78," the "architect" developer in this project.

Figure 2b illustrates a different structure, which we call *densely networked*. Instead of a single developer being responsible for integrating the whole project, now, this responsibility is evenly divided among a group of six different developers with a high degree of interdependence between them. There is no central "architect", but a group of developers interconnected. These densely networked projects are marked by a high degree of interdependence between different modules and developers, often approaching a "fully connected" state in which each developer depends on the code of each other developer. The degree of participation may vary (it is rare for all members of the project to contribute equally, and a set of primary developers normally emerges), but they cannot be easily distinguished in terms of their particular roles and responsibilities as developers.

Finally, Figure 2c shows a variation of the previous structures where not only a core of seven developers strongly connected can be found, but also a medium sized set of other four developers in the periphery of the project, that is, whose code does not interact. In this case, called the *core and periphery* division, a core phalanx of major developers are surrounded by a peripheral set of developers, less strongly connected. Note, again, that this is not a distinction between degrees of participation, but between forms of participation, as characterized by the interdependencies of the work. This is not an arrangement where a core group of developers is doing the majority of the work; rather, it is an arrangement where a core set of developers generate code that is strongly interdependent, while a peripheral set of developers tend to be more isolated from each other.

4.2 Forms of Peripheral Participation

We can further distinguish between various forms of peripheral participation. By tracing dependencies, we can see whether peripheral members are dependent on core members, or vice versa. Clearly, in some cases, the dependencies are mutual; these often characterize a peripheral developer who is playing a traditional role in the project, yet tends to be responsible for only some small portion of the system. More interesting, perhaps, are peripheral participants whose connection to the core is a one-way dependency; either core modules depend on peripheral ones, or peripheral ones on core modules.

Dependency, in our case, is a call from one component to another (or form the components of one developer to those of another.) So, peripheral modules that are called from core modules is a structure that is often associated with plugins, extensible component-based systems, or other systems in similar styles. In this case, a peripheral developer might develop a relatively self-contained module, which must be activated from the system core. We typically see, then, that the core developers, whose code is tightly interdependent, are associated with central functionality; the plug-in or self-contained module is peripheral in both functionality and in connectedness.

The inverse relationship characterizes a peripheral developer whose peripheral relationship is one of dependence on core functionality. Most commonly, we find this when a developer writes a test case, a novel user interface or application, or some other “wrapper” function that calls or relies upon the functionality of the rest of the system.

4.3 Core/Periphery Shifts

Earlier discussions of core and periphery focus on static structure, but we are interested in the dynamics of software processes, and in how participation shifts between core and peripheral participation. This phenomenon has been classified as both a learning and a political process, where one has to identify allies that back up a developer, “just like a statement in a scientific paper when it is accompanied by a large number of references and citations” [14].

This shift can be observed by examining the same open-source project at two different moments. By looking at the dependency structures in the source code, we can identify a developer’s contributions and their impact. In a shift from the periphery to the core, we expect to identify developers who initially contribute code that performs some function by calling others’ code. When these developers become more and more important in the project, their code starts to be called by other developers.

Figure 3 illustrates this in the project Megamek[1]. Initially, developer Hawkprime was located on the edge of the project, as measured by connections in the network. At left, he is connected to one other developer through his code (indicated by the directions of the interdependencies edges: from Hawkprime to the other). The reason for this is that BMazur is the principal interface author, consequently more central than Hawkprime. Later (right), Hawkprime assumes a more central role in the project. Now, he is also a *source* of dependencies because he is the author of an interface being implemented by others; now, other developers depend on his work. Furthermore, instead of only being connected to one other developer, he is now connected to six of them. Again, the shift can be noticed based on the relative *importance* of the code being contributed.

Using a similar approach, we identified the opposite effect, a developer’s shift from the core to the periphery of another project, ANT[2]. This time, the developer Umagesh initially had a central participation in the project. This can be observed by the five edges directed to him in the graph. Later, Umagesh shifted to the periphery of the project (Figure 4).

As in previous examples, the important issue here is not so much that these shifts take place; the movement of people between peripheral and central positions is both common-sense and empirically well observed. The important issue is the way in which it can be found in the data record; that is, that the pattern of participation is manifest within the inscription, and can be analyzed structurally through dependency analysis of the software artifacts.

4.4 Authorship Changes

One of the arguably factors leading to open-source success is the freedom in allowing developers to join and leave open source projects. Some authors indeed use the term active developers to indicate the developers who have contributed to the project in a specified time period. Of course, authorship information extracted from the configuration management repository will provide this information for those interested. Figure 5 shows these transitions for two separate projects. This figure displays a bipartite graph where square nodes are authors and the slim, oval nodes are files. Author nodes are never connected directly to one another but instead are connected through their relationship to shared source files. The file node, then, becomes the link between author nodes.

Figure 5a (left), this project (sugarcrm[3]) relied on a few authors who implemented nearly all components in the system. More recently, however, work has been split among five or six different authors, as indicated by the different colors in the right side of Figure 5a. That is, code in the system initially developed by one author has shifted ownership over time to other authors.

Adopting the same approach, we identified an author domination effect, where the code starts out owned by multiple authors and then a developer begins to take over (Figure 5b). In this case, the green and yellow authors’ code has begun to pervade nearly every aspect of two separate sub-modules in the project parrot[4].

4.5 Patterns of Stability and Changes

Finally, our last observation is with regards to patterns of stability and change in open source projects. For this example, we use data from the Python[5] project to explore stability and change within the context of the file tree structure of the project. Files and packages are nodes in this graph colored by authors. They are linked by containment relationships (between a package and a file). There is no connection between two files.

These graphs show how the structure of the source code (its organization in packages) is being used to structure the activities of the developers. For instance, Figure 6A describes a particular part of the source code initially implemented by a single developer. Later in the project, Figure 7A, it is possible to note that this developer remains the sole author of that module. That is, there was no change in the authorship of that code from one time snapshot to another. In contrast, part B of Figures 6 and 7 reveal how the authorship of code changes over time. Initially, in Figure 6B, the highlighted section was primarily authored by one author, but overtime, this section was distributed amongst many authors.

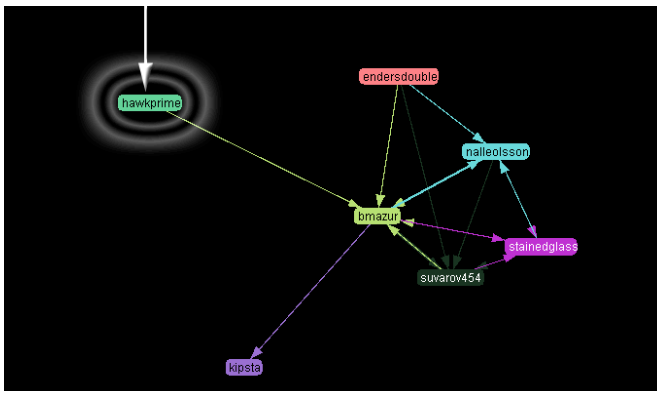
[1] http://megamek.sourceforge.net

[2] http://ant.apache.org/

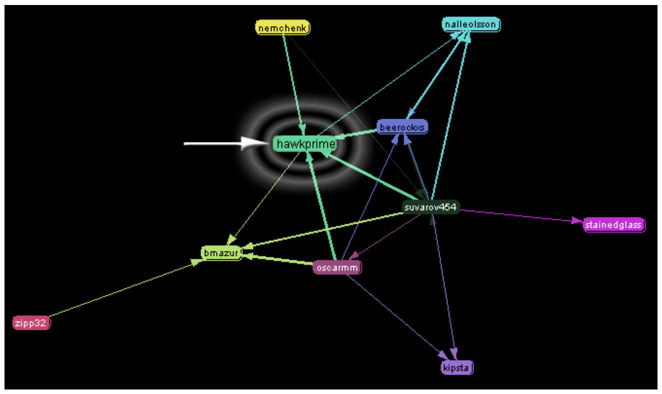
[3] http://sugarcrm.sourceforge.net

[4] http://cvs.perl.org

[5] http://www.python.org/

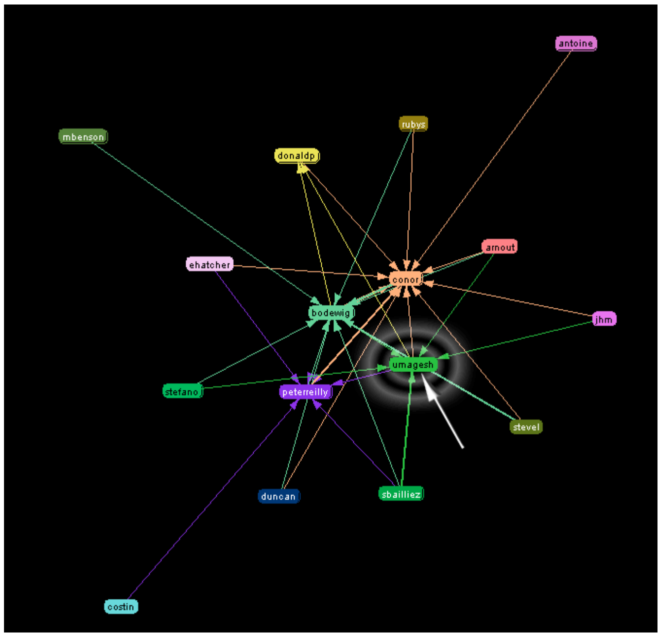


(a) “Hawkprime” in periphery

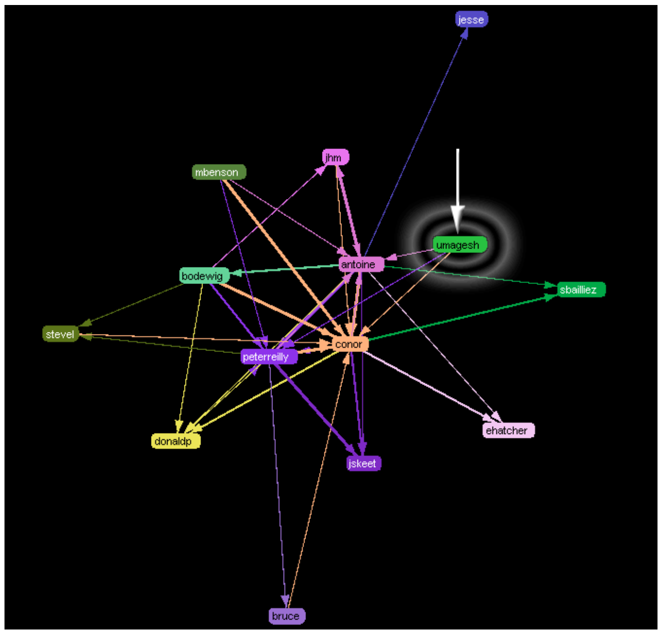


(b) “Hawkprime” shifts to core

Figure 3: Shift from periphery to core

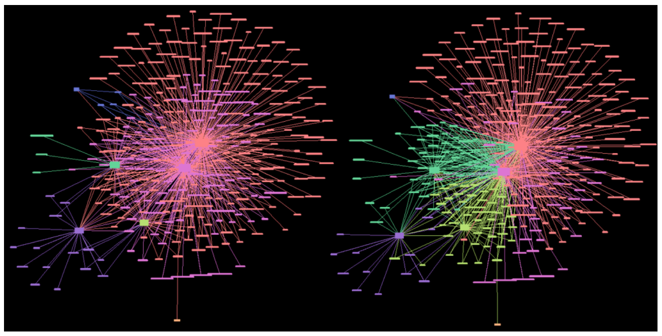


(a) “Umagesh” in core

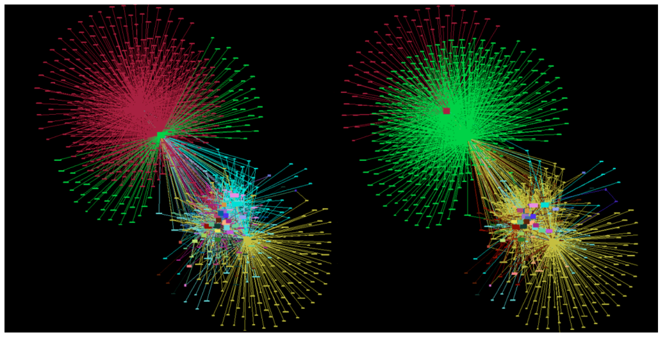


(b) “Umagesh” shifts to periphery

Figure 4: Shift from core to periphery



(a) Code ownership expands



(b) Code ownership contracts

Figure 5: Authorship changes in projects over time.

5. DISCUSSION

Augur provides us with a way to place software modules and software developers within the same frame of reference, and describe their relationships. By analyzing dependencies and activities, it highlights not just the links between people and code, but the links between people and others *through* code, and vice versa. This homogeneous analytic perspective is reminiscent of the actor-network approach [6, 7, 8, 23]. Actor-Network Theory maintains a deliberate agnosticism as to sources of agency, and insists that human and technological "actants" be given analytic parity. Latour [23] points out that functions embedded in social settings may also be delegated to technology; for instance, rather than have a policeman monitor road traffic to ensure conformance with speed limits, we can lay down "speed bumps" to achieve the same effect. (In the UK, speed bumps are sometimes known as "sleeping policemen," in vivid testimony to the potential relationship between physical and social elements.) Technological arrangements, as much as social arrangements, can be used to produce control and conformance with social norms. Actor Networks, then, bring together heterogeneous elements, including technologies, artifacts, and people.

Latour and Woolgar discuss the social processes that shape scientific practice and discourse [24]. Scientific processes, he suggests, are a means to "delete modalities," that is, to remove the conditions on truth statements. So, through this process, statements of the form "in the fourth experimental run, a correlation was observed between inputs and outputs" might be transformed into "our research suggests that outputs are proportional to inputs", which in turn can be transformed into "O = k.I"; at each stage, some of the conditional elements are removed, and more universal statements can be made. Part of successful scientific practice, then, is the construction of networks that can help to "stabilize" particular results, deleting modalities by establishing the reliability of observations, results and conclusions. In this view, prestigious institutional affiliations, sensitive laboratory equipment, experimental verification, and solid theoretical foundations are not simply historical or technical; they are elements in the network, playing a strategic role in the stabilization of scientific facts.

One concept arising from this perspective on scientific processes is that of the "obligatory passage point" – a narrowing of the network that designates some particular element as one that must be navigated in order to achieve a result. As befits the homogeneous treatment of heterogeneous elements in actor-network theory, this might be any sort of entity. Professional certification might play such a role, for example; so might a particular theory, a scientific leader, a particular laboratory, and so forth. We can see how this can operate in open source domains.

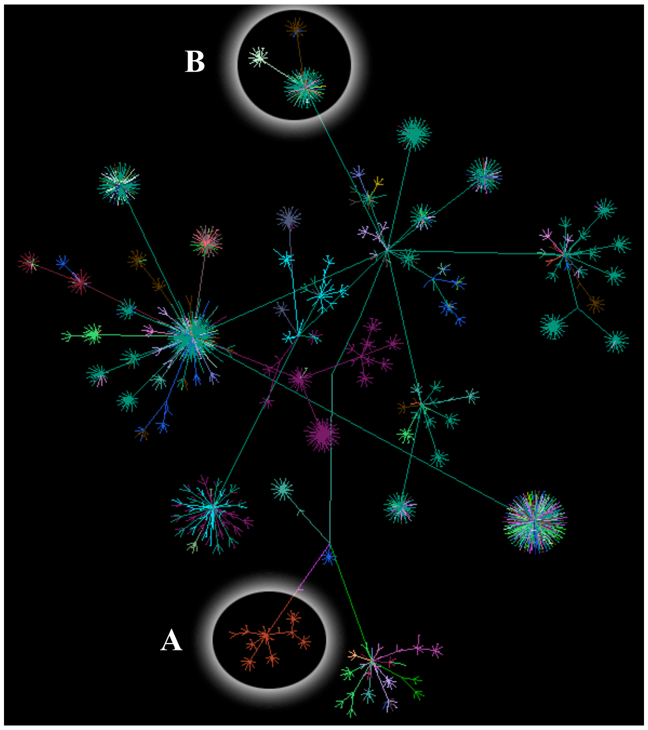


Figure 6 - Stability and Changes - First Moment

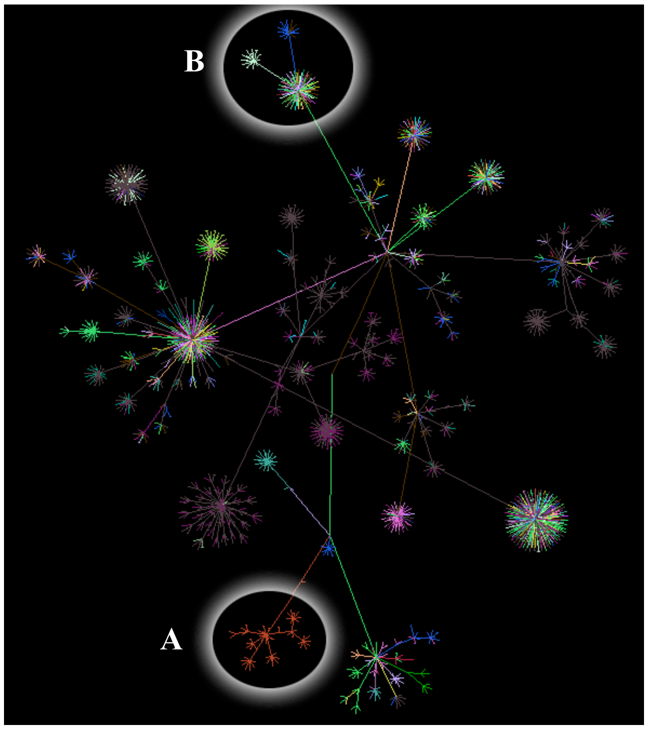


Figure 7 - Stability and Changes - Second Moment

In general, the relevance of these concepts to the work presented here is the light that they cast on the interplay between social and technical in distributed activities. A number of authors have explored aspects of the social structure of open source projects [10, 27, 28, 36]. Our approach has been to look at the ways in which aspects of the social and organizational structure are both inscribed into and achieved through the technological organization of the underlying artifact, the software source code. The central lesson here is two-fold. First, that while the rhetoric of open source is of openness and access, the practice of open source is about closedness and regulation; essentially, a central consideration in managing a successful technical project is to ensure the consistency and quality of the technological artifact under production, which is managed by vetting both contributors and contributions, and so a structure must be produced by which such a vetting can be achieved. Second, this structure is manifest collectively by the social and technical organization of the project. The same engineering principles by which software systems can be organized to achieve technical properties (modularity, extensibility robustness, etc.) are also ones by which activities can be partitioned and managed, and access to the system limited. What we see in these examples, then, is essentially the emergence of obligatory passage points within software development practices. Those points may be technical or human elements. As presented among the forms of peripheral participation, there may be a particular module or component into which others must be hooked; a dispatch table, an event loop, or so forth.

That these structures should emerge in successful software projects is not surprising; these projects, after all, require careful coordination, and some mechanisms are needed to ensure that this takes place. That they should emerge within open source projects, while not surprising, is nonetheless interesting, in light of the open source movement's focus on participation and accessibility. What is particularly interesting, though, is that these processual elements of software production can, themselves, be found within the software structures that are the focus of activity. While Latour and others argue that processes and social structures are inscribed into scientific and technical artifacts, our experiences with Augur point to the ways in which, for software artifacts, they might be "read off" again. Our empirical examinations demonstrate that both software components (modules) and software developers can act, for example as obligatory points of passage; the structure of a software project both reflects and constrains the development process. An important piece of further work concerns the automatic recognition and extraction of these patterns; our work was oriented first towards determining whether process patterns could be found within software repositories. The answer is yes.

6. CONCLUSIONS

Distributed software development presents two sources of complexity to its participants – the complexity of the software artifacts under development, and the complexity of the process of developing those artifacts. We have presented a study of software artifacts, conducted using a visualization tool, which demonstrates how these twin sources of complexity are intertwined. Software artifacts are not merely the objects of software development processes, but are also the means by which those processes are enacted and regulated. The structure of the artifact both reflects the processes by which it has been created and can be used to control those processes by centralizing points of access, by regulating the relationships between independent activities, and by making visible the relationships between individuals. It is a means, then, by which the articulation work of the project can be carried out [30].

The intertwining of artifacts and activities is no surprise to CSCW researchers, of course. What is of interest here is to see how it happens in one particular case. Free and open source development is a particularly enlightening domain within which to study these concerns. On a mundane level, the artifacts of open source development are easily available; but more significantly, the inter- or extra-organizational context of much open source development means that the tools of the trade – CM systems, web sites, and the source code itself – are the site at which access and activity structure are negotiated. In particular, we have shown how both individuals and software components may act as "obligatory passage points," constrictions in the loose network of artifacts and activities that can be used to achieve local and partial stabilizations of dynamics socio-technical settings. Further, we have shown the use of computational tools to help make these structures visible.

Our approach has been methodologically unusual, since we have been conducting, essentially, an "archeology" of software development processes. The critical next step is a more immersive engagement with large-scale distributed software development enterprises, in order to gain a better understanding of these processes "close up." These open source settings provide a valuable site for examining the evolution of practice around technological artifacts – a central consideration for CSCW. Our explorations demonstrate that software artifacts can reveal the relationship between technical and social structure of large-scale development projects, and so suggest that collaborative tools can exploit not only technical but also social structures in supporting collaborative software development.

7. ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under awards IIS-0133749, IIS-0205724 and IIS-0326105, by Intel Research, and by the Brazilian Government under CAPES grant BEX 1312/99-5.

8. REFERENCES

[1] Adler, P. (2003). Practice and Process: The socialization of software development. Unpublished manuscript, University of Southern California.

[2] Bannon, L. & Bødker, S. 1999. Constructing Common Information Spaces. Proceedings of European Conf. Computer-Supported Cooperative Work ECSCW97, 81-96.

[3] Bowers, J. 1992. The Politics of Formalism. In Lea (ed), Contexts of Computer-Mediated Communication, 231-261. Harvester Wheatsheaf.

[4] Bowker, G. and Star, S.L. 1997. Sorting Things Out: Classification and its Consequences. Cambridge, MA: MIT Press.

[5] Brooks, R. 1983. Towards a Theory of the Comprehension of Computer Programs. Intl. Jnl. Man-Machine Studies, 18, 543-554.

[6] Callon, M. 1986. Some elements of a sociology of translation: Domestication of the scallops and fishermen of St. Brieuc Bay. In Law (ed.), Power, Action and Belief: a new sociology of knowledge?, 196-233. London: Routledge.

[7] Callon, M. 1986. The Sociology of an Actor-Network: The case of the electric vehicle. In Callon, Law, and Rip, (eds.), Mapping the Dynamics of Science and Technology, 19-34. London: Macmillan.

[8] Callon, M. 1991. Techno-Economic Networks and Irreversability. In Law (ed.), A Sociology of Monsters: Essays on Power, Technology and Domination, 132-161.

[9] Conway, M. E. (1968). "How Do Committees invent?" Datamation 14(4): 28-31.

[10] Crowston, K. and Howison, J. 2004. The Social Structure of Free and Open Source Software. First Monday, 10(2), 2005.

[11] Davis, S. 1990. The Nature and Development of Programming Plans. Intl. Jnl Man-Machine Studies, 32(4), 461-481.

[12] de Souza, C., Redmiles, D., and Dourish, P. 2003. Breaking the Code: Moving between Private and Public Work in Collaborative Software Development. In Proceedings of the ACM Conference on Supporting Group Work GROUP 2003.

[13] de Souza, C.R.B., Redmiles, D., Cheng, L.-T., Millen, D. and Patterson, J., Sometimes You Need to See Through Walls - A Field Study of Application Programming Interfaces. In Proceedings of the ACM *Conference on Computer-Supported Cooperative Work (CSCW '04)*, (Chicago, IL, USA, 2004), ACM Press, 63-71.

[14] Ducheneaut, N. "The reproduction of Open Source software programming communities." Unpublished Ph.D. thesis, U.C. Berkeley.

[15] Eick, S., Steffan, J., and Sumner, E. 1992. Seesoft: A Tool for Visualizing Line-Oriented Software Statistics. IEEE Trans. Software Engineering, 18(11), 957-968.

[16] Froehlich, J. and Dourish, P. 2004. Unifying Artifacts and Activities in a Visual Tool for Distributed Software Developers. Proc. Intl. Conf. Software Engineering ICSE 2003 (Edinburgh, Scotland). New York: IEEE.

[17] Fujimura, J. 1987. Constructing "Do-Able" Problems in Cancer Research: Articulating Alignment. Social Studies of Science, 17(2), 257-293.

[18] Fujimura, J. 1997. The Molecular Biological Bandwagon in Cancer Research. In Strauss and Corbin (eds), Grounded Theory in Practice, 131-145.

[19] Grinter, R. 1995. Using a Configuration Management Tool to Coordinate Software Development. Proc. ACM Conf. Organizational Computing Systems COOCS '95 (San Jose, California), 168-177. New York: ACM.

[20] Grinter, R. 1998. Recomposition: Putting it All Together Again. Proc. ACM Conf. Computer-Supported Cooperative Work CSCW'98 (Seattle, WA).

[21] Herbsleb, J. and Grinter, R. 1999. Splutting the Organization and Integrating the Code: Conway's Law Revisited. Intl. Conf. Software Engineering ICSE (Los Angeles, CA), 85-95.

[22] Herbsleb, J., Mockus, A., Finholt, T. and Grinter, R. 2000. Distance, Dependencies, and Delay in a Global Collaboration. Proc. ACM Conf. Computer-Supported Cooperative Work CSCW 2000 (Pittsburgh, PA), 319-328.

[23] Latour, B. 1994. Where are the missing masses? The sociology of a few mundane artifacts. In Bijker and Law (eds.), Shaping Technology / Building Society: Studies in Sociotechnical Change, 225-258. Cambridge, MA: MIT Press.

[24] Latour, B. and Woolgar, S. 1979. Laboratory Life: The Social Construction of Scientific Facts. Beverly Hills, CA: Sage.

[25] Lynch, M. 1985. Discipline and the Material Form of Images: An Analysis of Scientific Visibility. Social Studies of Science, 15(1), 37-66.

[26] Lynch, M. 1988. The Externalized Retina: Selection and Mathematization in the Visual Documentation of Objects in the Life Sciences. In Lynch and Woolgar (eds), Representation in Scientific Practice. Cambridge, MA: MIT Press.

[27] Mockus, A.,, Fielding, R. and Herbsleb, J. 2000. A Case Study of Open Source Software Development: The Apache Server. Intl. Conf. Software Engineering (Limerick, Ireland), 263-272.

[28] O'Mahoney, S. and Ferraro, F. 2004. Managing the Boundary of an 'Open' Project. Harvard NOM Working paper No. 03-60. Cambridge, MA: Harvard Business School.

[29] Parnas, D. L. (1972). On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM 15(12): 1053-1058.

[30] Schmidt, K. and Bannon, L. 1992. Taking CSCW Seriously: Supporting Articulation Work. Computer-Supported Cooperative Work, 1(1-2), 7-40.

[31] Schmidt, K and Wagner, I. 2004. Ordering systems: Coordinative practices and artifacts in architectural design and planning. Computer Supported Cooperative Work, 13 (5-6), 349-408.

[32] Sharrock, W. and Button, G. 1997. Engineering Investigations: Practical Sociological Reasoning in the Work of Engineers. In Bowker, Star, Turner, and Gasser (eds), Social Science, Technical Systems, and Cooperative Work: Beyond the Great Divide, 79-104. Mahwah, NJ: Lawrence Erlbaum.

[33] Star, S.L. and Ruhleder, K. 1994. Steps Towards an Ecology of Infrastructure. Proc. ACM Conf. Computer-Supported Cooperative Work CSCW'94 (Chapel Hill, NC), 253-264. New York: ACM.

[34] Suchman, L. 1983. Office Procedures as Practical Action: Models of Work and System Design. ACM Trans. Office Information Systems, 1(4), 320-328.

[35] Wagner, D. and Dean, D. 2001. Intrusion Detection via Static Analysis. Proc. IEEE Symposium on Security and Privacy (Oakland, CA).

[36] West, J. and O'Mahoney, S. 2005. Contrasting Community Building in Sponsored and Community Founded Open Source Projects. Proc. Hawaii Intl. Conf. Systems Sciences HICSS-3.