From Personal Tool to Community Resource: What's the Extra Work and Who Will Do It?

Erik H. Trainer, Chalalai Chaihirunkarn, Arun Kalyanasundaram, James D. Herbsleb
Institute for Software Research
Carnegie Mellon University
5000 Forbes Ave., Pittsburgh, PA 15213
{etrainer, cchaihir, arunkaly, jdh}@cs.cmu.edu

ABSTRACT
Sharing scientific data, software, and instruments is becoming increasingly common as science moves toward large-scale, distributed collaborations. Sharing these resources requires extra work to make them generally useful. Although we know much about the extra work associated with sharing *data*, we know little about the work associated with sharing contributions to *software*, even though software is of vital importance to nearly every scientific result. This paper presents a qualitative, interview-based study of the extra work that developers and end users of scientific software undertake. Our findings indicate that they conduct a rich set of extra work around community management, code maintenance, education and training, developer-user interaction, and foreseeing user needs. We identify several conditions under which they are likely to do this work, as well as design principles that can facilitate it. Our results have important implications for future empirical studies as well as funding policy.

Author Keywords
Software sharing; scientific software; extra work; scientific communities; qualitative methods.

ACM Classification Keywords
H.5.3 [Information Interfaces and Presentation (e.g., HCI)]: Group and Organization Interfaces – Computer supported cooperative work.

INTRODUCTION
We are in an era of distributed, large-scale science that depends not only on new technological advances, but also on "human infrastructure," complex arrangements of people, organizations and communities [22]. Studies of cyberinfrastructure and e-Science have pointed out that creating this infrastructure is an ongoing task, involving continuous collaboration, alignment, and adjustment among many stakeholders [3]. Sharing scientific resources, a central dimension of this collaboration, has been a topic of great interest to the CSCW community as of late [4,10,29].

Sharing scientific resources requires *extra work*, voluntary or involuntary unpaid labor, to make them generally useful. A significant portion of recent research on sharing in science has documented extra work from the perspective of the recipient: locating data [39], interpreting data [5,29], and assessing their reliability [10,15]. There is also extra work from the perspective of the sharer. Standardizing data, for instance, requires reaching agreement on key terms, creating protocols for data collection, providing tutorials and training on implementing the standards, and translating the standards to forms, spreadsheet templates, and interfaces [2,26].

The above studies have focused primarily on the work scientists do to share *data*. Other than work showing that scientists share software based on social ties [18], we know little about the extra work to make contributions to *software* generally useful, even though software is a major output of the scientific process. More than other assets produced by scientific work, such as published results and archived data, software requires continuous maintenance effort or it soon loses its value, as it becomes incompatible with new releases of operating systems, middleware, and complementary software.

Extra work provides powerful leverage for other scientists who can use the software to produce new knowledge. If a scientist spends 10 hours writing a piece of software, plus an additional hour to make it generally useful, other scientists who use it get the benefit of that 10 hours of work. Even if only 1 other scientist uses it, the benefit from the perspective of the community is 10 hours. If 2 scientists use it, 20 hours, and so on. If we can understand what this extra work is, and under what conditions it is likely to happen, we can find ways to facilitate it.

We therefore address the following research questions:

(1) *What are the kinds of extra work that scientists do to make contributions to software generally useful?*

(2) *Under what conditions are scientists likely to perform (or not perform) extra work?*

(3) *What are the design principles that can facilitate extra work?*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
CSCW '15, March 14 - 18 2015, Vancouver, BC, Canada
Copyright 2014 ACM 978-1-4503-2922-4/15/03…$15.00
http://dx.doi.org/10.1145/2675133.2675172

To answer these questions, we conducted a multiple case study of four scientific software communities. We interviewed developers and end users, having them describe contributions to and use of the software, and activities within their communities. We found that our participants conducted a rich set of extra work, and several considerations influenced their decisions to do so. We also identified several principles that can facilitate extra work. In the following sections we review related research, describe our study, present the results of our interviews, and discuss the implications of our findings.

BACKGROUND

The Human Work of Large-Scale Science

The research challenges of today demand large-scale collaborations involving multiple research institutions and scientific domains, often as a condition of funding. Creating the necessary infrastructure for such projects is an ongoing task requiring the establishment and maintenance of people, groups, and technologies. For instance, Bietz et al. [3,4] talk about "aligning," which is the work that developers do to enact a relationship in a way that enables it to produce within a cyberinfrastructure. Developers of middleware, for example, must balance building systems that can be used across multiple projects, while also ensuring that their software meets the needs of funding sources. In order to find potential collaborators, developers attend seminars, watch funding calls from various agencies, and follow trends in multiple scientific domains [4].

Another kind of work, "leveraging," emphasizes how an existing relationship with a person, artifact, or organization can build or strengthen another relationship [3]. To get access to a novel technology, a project may create a partnership with a research project located at the same university that is developing the technology.

Extra Work of Sharing Scientific Resources

In the context of large-scale scientific collaboration, recent research has examined the work of sharing scientific data, not scientific software. A notable exception is work by Huang et al. [18] that identified social practices around the sharing of and control over commercial and open-source software in a bioinformatics research team. For instance, in order to use software that another team developed, scientists leveraged existing relationships with members of the other team and negotiated co-authorship on publications. Our work is distinct by looking at the work to make features of the software generally useful to a community of users, beyond a particular laboratory, and on open-source software in particular.

Much of the research on data has documented work required by the recipient. Zimmerman [39], for example, documented how ecologists locate data for reuse. She found that ecologists devised systematic sampling methods and used research journals to bound their data searches by timeframe and geography.

Other research has focused on the work of data interpretation [5,29]. For instance, Rolland & Lee [29] found that postdoctoral researchers reusing cancer epidemiology datasets spent significant amounts of time consulting manuscripts to see how the data were written about, reading study protocols and documentation to understand how key variables were constructed, and even tracking down former study members in their new jobs.

Recipients of scientific data also assess the data's relevance and trustworthiness. Faniel & Jacobsen [10] observed that earthquake engineering researchers assess relevance by generating criteria related to their research questions and comparing those criteria to colleagues' experimental test setups and parameters, which they find in journal articles describing the experiments or through direct contact with the researchers. To assess trustworthiness, they consult experiment documentation first and speak with colleagues who produce the data to get basic facts or clarify inconsistencies in the data.

Sharing scientific resources also requires extra effort on the part of the sharer. For example, standardizing data for sharing can involve creating laboratory protocols, reaching agreement on definitions of key terms, translating standards to forms, templates, smartphone apps, and web sites, and developing material transfer agreements [2]. In a study of a standardization process in an ecological research community, Millerand et al. [26] documented how developers of the standards created training sessions for information managers at distant sites to teach them how to implement the standard.

There are likely some kinds of extra work that the sharer is in a better position to do, and some kinds of extra work that the recipient is in a better position to do. On the one hand, when deciding to offer up a piece of software for the community, the sharer understands what its functions do, and will be able to modify it more easily than the recipient. It will therefore be appropriate for that scientist to create documentation for the software and fix bugs that surface after it goes into operation. On the other hand, the recipient may know more about intended use cases for the software, or other data and tools with which the software must interoperate. The recipient can also provide the sharer with valuable feedback. The role of the recipient is an important one. In fact, product innovations often come from users, who know more about what the product should do, and what it means to do it better [13,14].

It is clear why recipients do extra work; they are the ones who need and will use what sharers provide. The motivations of sharers, however, are not as easily explained. Undertaking extra work without tangible reward is a very real concern, shown in early CSCW research to be a reason why groupware systems fail [12,27]. Sharers of data may nevertheless get fuller use out of their data, or the ability to negotiate co-authorship on publications produced using the data. To understand why someone would do extra work on software, we turn to the literature on open-source software.

Open-Source Software: Extra Work for Others (for free)

There are three primary individual motivations for contributing to open-source software [8]. *Intrinsic* motivations include fun and enjoyment [21] and learning opportunities [28,31,38]. *Internalized extrinsic motivations* include fulfilling personal needs [20,23].

Extrinsic motivations such as reputation and opportunities for career development have been the focus of increased study in recent years [9,25,28]. Studies of transparent software development environments such as GitHub (http://github.com/) have found that developers actively manage their profiles to gain greater attention and visibility for their work [24]. The number of followers a developer has, for example, signals their status. The transparency into development activities that GitHub provides allows others to gauge developers' expertise [9]. Employers have even started using GitHub to find and recruit potential employees [25].

The trouble is that whereas reputation is an effective motivator in the world of open-source, it is significantly less of a motivator—perhaps even irrelevant—in the world of scientific software. The reason is that scientists receive reputation for the results they publish, not the software they develop. Scientists receive credit for software only indirectly. In a study of BLAST [1], a key bioinformatics tool, Howison & Herbsleb [16,17] concluded that improvements to the software motivated by academic credit were less likely to be integrated. They argued that this is because integration makes it harder to see who did what, undermining the ability of reputation to function as a reward.

METHOD

To address our research questions, we conducted a multiple case study of four scientific software communities characterized by the availability of rich data and community activity. Three considerations influenced our selection of cases. First, we wanted clearly infrastructural projects and end user projects, both in the same domain, in order to understand how the presence of close ties to users influences the kinds of extra work conducted. We found very active bioinformatics projects meeting this criterion in Biopython and Bioclipse. Biopython is a set of Python software libraries designed for use by other scientists who develop software [6]. The project began in 1999, and as of this writing has 69 contributors to the source-code, and 487 "stargazers," people who subscribe to updates made to the repository, on GitHub. The source-code has been "forked," cloned by someone to use it as a starting point for their own contributions, 283 times. Bioclipse, in contrast, is a biological workbench application providing a range of functionality for end user biologists [33,34]. The first release of Bioclipse was in 2007. In 2009, Bioclipse was re-written to allow end users to automate functionality using the Bioclipse Scripting Language, and to share reproducible scripts. The project has 8 contributors, 13 stargazers, and 11 forks on GitHub.

Second, we wanted to look at a project that has paid developers. The existence of a paid core might influence the extra work that volunteers are willing to do, in that volunteers may be less willing to do work that they perceive to be the responsibility of paid developers. Throughout this paper we also use the term "core" to refer to developers who are listed as such on their community's website, who self identify as core, or who others identify as core. All paid developers are core, but not all core are paid.

We found a project with paid developers in Bioconductor, which comprises components, often R packages, for analysis of genomic data generated by biological wet lab experiments [11]. First released in 2001 with 15 packages, Bioconductor now contains 824 packages. Bioconductor's source-code is hosted in a Subversion repository instead of GitHub, so we were unable to collect information about contributors, stargazers, and forks. The Bioconductor website, however, tracks download statistics for all packages. From August 2013 to July 2014, there were 9,354,646 downloads from 2,35,401 IP addresses (http://www.bioconductor.org/packages/stats/). In addition, the Bioconductor mailing list has 3,500 subscribers, suggesting a large community of users. Each package has one or more volunteers who maintain it (although paid developers may also maintain packages). The team of paid programmers is located at the Fred Hutchinson Cancer Research Center in Seattle, Washington, USA. Their job is to review user-submitted components, manage official releases, provide user support, and develop experimental packages.

Third, we wanted another project to contrast with the others on many dimensions, with the intention to see large differences and to see if our findings generalized across these many dimensions. We therefore selected NetLogo, a project used for simulation, instead of analysis [37]. Developed in 1999, NetLogo was closed source until it was moved to GitHub in 2011, where it has 13 contributors, 165 stargazers, and 46 forks. NetLogo is in use by educators as well as researchers in the physical and social sciences. NetLogo end users are also active contributors, adding another dimension to sharing in this particular community. Although they do not make direct source-code contributions to NetLogo, end users do write models using the NetLogo language that simulate various natural and social phenomena, and can share these models with others on the Modeling Commons (http://modelingcommons.org/), a community model sharing site. The Modeling Commons currently lists 1,272 models. End users can also write and share extensions, which are separate plug-ins that add capabilities to the modeling environment. For instance, the R extension (http://netlogo-r-ext.berlios.de/) adds the capability to send data between NetLogo and R and to call

R functions from within NetLogo. The NetLogo extensions page lists 37 user submitted extensions. Like Bioconductor, NetLogo has paid developers.

Among the three biology-related projects in our sample, Bioconductor and Biopython appear to be more popular and actively developed compared with Bioclipse. Using a tool's Google Scholar search results as a rough indicator of its usage in scientific work, we found that 24,800 publications mention Bioconductor, 1,420 mention Biopython, and 263 mention Bioclipse. NetLogo appears to be popular across multiple domains, and the simulation environment, models, and extensions are all actively developed. We found that 8,970 publications mention NetLogo.

Data Collection

We conducted semi-structured interviews with 39 participants (see Table 1). We identified our participants by inspecting a combination of community web pages and wikis, mailing lists, and commit access listings from the source code repositories. For Bioconductor and NetLogo, we targeted participants who were paid developers and participants who contributed in their spare time in order to understand differences in motivations and types of extra work these two groups conducted. For NetLogo, we targeted equally model and extension developers in order to understand differences between extra work associated with each of these end user-submitted tools and the main NetLogo source-code. We solicited participants by e-mail and interviewed them using either Skype or Google Hangouts. In the interviews we first asked participants to talk about their role and activity in the project in general, and then we asked about their recent contributions in greater detail. Interviews lasted 45 minutes on average. A transcription service firm transcribed all interviews.

Analysis

We used techniques from grounded theory [7] to analyze the interview transcripts. We first imported all transcripts into the Dedoose qualitative data analysis software package [32]. We then identified and conducted open coding on statements about work that participants carried out to make the software of general use, and their reasons for doing so. When possible, we triangulated participants' statements with work artifacts hosted in project source-code repositories as well as descriptions of, and discussions about, those artifacts posted in mailing lists and web pages.

In the next phase of analysis we wrote, shared, and discussed descriptive memos about emerging themes in the dataset. We met weekly to unify, refine and collapse codes where there was commonality, using themes from our memos as support. We applied the resulting set of codes to the remaining interviews, sometimes revealing additional behaviors not captured by our existing set. In such cases we extended our codes as appropriate. We continued this process until our data no longer revealed new phenomena captured by our categories.

	Volunteer	**Paid**
Biopython	P1, P2, P3, P4, P5, P6, P7, P8, P21, P24, P30, P33, P40	
Bioclipse	P17, P18, P22, P26, P29	
Bioconductor	P31, P35, P37, P38, P44	P23, P27, P41, P42
NetLogo	P11, P12, P13, P14, P15, P16, P20, P25, P36	P9, P10, P34

Table 1. Summary of interview participants.

RESULTS

RQ 1: What are the kinds of extra work that scientists do to make contributions to software generally useful?

We found that participants conducted a rich set of extra work. Our analysis revealed five categories of extra work: *community management*, *code maintenance*, *education and training*, *developer/user interaction*, and *foreseeing user needs*.

Community Management

Participants described doing work that supported community management activities. This work included following community norms and guidelines, attending core team meetings, evaluating potential contributions, promoting the project in the broader community, soliciting contributions from community members, announcing work in progress, and staying aware of community members' activities.

Abiding by Community Norms and Guidelines

Community norms and guidelines provided participants guidance about what extra work to conduct. Participants from Biopython and Bioconductor (P37, P31, P40, P7, P2) described following guidelines, such as thoroughly testing the code, collating examples and documentation, reusing common data structures, following naming conventions, and breaking the code apart into logical pieces. For instance, upon failing the tests required to check his code into the Biopython repository, one respondent described that a core developer told him to change the syntax of his code to make it compatible with all the versions of Python that Biopython supports (P40).

The Bioconductor community has additional expectations about the work that follows code contribution. Members of the paid core (P23, P27, P41, P42) explained that each Bioconductor package has one or more maintainers who address support requests and bug reports.

NetLogo and Bioclipse, in contrast, do not have clear guidelines with respect to contribution. Yet NetLogo end users who shared their models, for instance, learned how to structure those models based on reviewing existing models bundled with the NetLogo software and models shared on the Modeling Commons (P11, P13, P16). From these reviews, participants sensed that models should be simple and generalizable:

"What people tend to like are models that are relatively simple, and if you want something else, you put them in a separate model. [My model] on the [Modeling Commons] isn't exactly like that, but the one that's on my hard drive is not at all like that. It's just a bunch of different tools thrown into the same NetLogo model…no one wants to look at that unless they're doing stuff with it." (P11, NetLogo)

P11 described spending a few hours over a week's time to clean out the extraneous functionality in his model before sharing it on the Modeling Commons. Another participant described that he believed his model was simple enough to be generally applicable, but that he still needed to document it, as suggested by the NetLogo model template, before he felt it was ready to share with others (P13).

Attending Core Team Meetings

Attending face-to-face meetings played an important role for core developers. Members of the Bioconductor paid core team, for example, described meeting every three weeks to discuss challenges related to the development of experimental packages, policies around package reviews, how to make resources on the website more usable, and developing courses on the use of particular Bioconductor packages (P23, P27). NetLogo core developers met face to face every six months to discuss design changes, demonstrate prototype functionality to one another, and discuss future directions for the project with their grant partners (P20, P10).

Evaluating Submitted Contributions

Participants also talked about the work of assessing potential contributions, often doing their best to help the submitter get the contribution into an acceptable state. This typically involved back and forth discussions in a GitHub "pull request," a workflow method where a contributor submits proposed code changes to a project, which are integrated into the main branch only with the repository owners' approval. Pull requests are public; anyone can comment on them. Respondents from projects hosted on GitHub (i.e., Biopython, Bioclipse, and NetLogo) recalled that pull requests generated discussion from repository owners and other developers, who pointed out possible incompatibilities with the code and other tools within their workflows, and suggested possible fixes and improvements (P10, P9, P6, P1, P7, P20, P42).

Paid developers of Bioconductor are responsible for reviewing user-submitted packages. As such, we did not consider their reviews extra work. However, we found that some individuals not paid to work on Bioconductor (e.g., P38) volunteered to review packages, which entailed reading through people's code, checking for adequate unit tests, checking for reuse of common data structures, and looking for thorough documentation. A paid Bioconductor developer explained that the review process usually took a whole afternoon to complete (P42).

Advertising Project, Demonstrating Impact

To promote their software and generate enthusiasm for it among the broader community, participants shared updates to the software on Twitter and Facebook (P10, P34, P44), wrote papers describing the software (P23), and organized workshops (P29, P23, P27).

Often underlying project promotion was motivation to increase the software's perceived scientific impact. Participants talked about the importance of demonstrating to funding agencies their software's value to the community, employing usage indicators to do so (P27, P22, P29, P34). For instance, in annual reports to funders, participants included the Bioconductor download statistics for their packages and citations to key publications (P22, P27). In fact, all projects in our sample provide specific citation guidelines for scientists who publish results obtained using the software.

Soliciting Contributions

Several participants talked about the work of finding project contributors (P7, P38, P34, P42). Upon noticing activities in end users' project "forks," participants e-mailed them asking when the code would be ready to integrate (P7, P8, P38). Some core developers (e.g., P34) also encouraged end users to implement ideas for features discussed on the mailing list.

Several core developers from Biopython and Bioconductor described having ideas for functionality, but prioritized other responsibilities over implementing those ideas (P7, P4, P41, P42). In response, they applied to the Google Summer of Code (GSoC) (https://developers.google.com/open-source/soc/) program to find a student to produce the code. GSoC is an annual program sponsored by Google that pays students to develop features for an open-source project during the summer. In some cases, students continued to contribute after the summer (P4, P5, P8, P44).

Announcing Work in Progress

Participants sometimes made announcements on the mailing list to notify others what they were working on. Participants described that they did this to gather suggestions for improvements (P36, P8, P5) as well as feedback on the usefulness of the functionality (P5). One participant from Biopython described that he preferred to announce proposed changes over the mailing list before making a pull request because the mailing list can potentially reach a broader audience—not just developers but end users as well (P5).

In the Biopython project in particular, GSoC students posted weekly blogs containing descriptions of their ongoing work (P8, P7, P30).

Monitoring other People's Activities

Participants also described trying to get a general sense of what others in the community were doing. To this end they browsed the project mailing lists, GitHub project activity visualization and feeds, and question and answer forums. For example, participants reported periodically checking active forks of their projects to get a general sense of what people were doing with their code (P9, P7, P34, P36). As the de facto leader of Biopython told us: "*I find that useful to know what other people have been up to just in case there's something I need to be aware of" (P7, Biopython).*

Code Maintenance

We found that participants made efforts to ensure that their software worked correctly before and after making it publically available. These efforts included fixing bugs, managing dependencies in the code, monitoring code usage, and testing.

Fixing Bugs

One of the most common types of extra work during maintenance was fixing bugs. Respondents described receiving bug reports through issue tracking systems and through e-mail, over the mailing list, and in pull requests themselves (P12, P37, P20, P24, P1, P35). In many cases, participants were willing to fix reported bugs. In other cases, decisions to fix bugs depended on the importance of the bug, the existence of workarounds, and developers' spare time (P10, P12).

Social capital and reciprocity seemed to play important roles in fixing bugs (P20, P29, P5, P14). For example, one participant recalled a NetLogo extension that he wrote years earlier but no longer used. He described fixing bugs that a former professor of his continued to report even though it was *"kind of a pain to work on it" (P20, NetLogo).* Members of the Bioclipse project contributed to another project on which Bioclipse has a dependency. One respondent explained that this *"giving and taking"* ensured that bugs in the other project were fixed quickly (P29, Bioclipse).

Managing Dependencies

We found that participants spent time resolving dependencies in their code. The most common occurrence was where a participant noticed that their software was failing. The participant would very quickly contact the developer of the depended upon code through e-mail or the mailing list to resolve the error (P37, P31).

Work associated with dependency management seemed to be linked to the architecture of the project. The Bioconductor package system, for example, allows contributors to reuse certain data structures and existing algorithms, enabling reproducible research and interoperability among packages (P27). It is exactly this reuse that creates dependencies. The Bioclipse and NetLogo architectures allow contributors to write extensions, which are like plug-ins, (and models in the case of NetLogo) without needing to know how to modify the core components. This allows extensions to evolve independently (P10), but puts the maintenance burden on extension developers (P10, P15). For example, one NetLogo extension developer maintained two different versions of his extension so that it worked with the older and newer versions of NetLogo (P15). Because Biopython is a set of libraries, it comprises many wrappers for other tools. Participants explained that they were accustomed to new changes in file formats and new switches arising from new functionality included in new releases of BLAST [1] (e.g., P7). Fortunately, very often the fix involved only changing a few lines of code.

Monitoring Code Usage

We found that participants actively kept track of how others used their code. Some participants occasionally checked the number of downloads of their software (e.g., P15, P2). Others, in addition, instrumented their software to collect usage data (P22). Participants described these statistics as important to include in grant reports and funding applications (P22, P27, P10).

Usage information also supported developers in addressing user problems. The core developers of NetLogo described setting up RSS feeds to watch for questions about NetLogo on Stack Overflow (http://www.stackoverflow.com/), a popular question and answer website for programmers (P10, P34). Maintainers of Bioconductor packages watched the mailing list for questions about their components (P31, P44). Actively monitoring how others were using code was relatively rare in Biopython. One respondent mentioned setting up a RSS feed to monitor questions about his contributions. He also set up a special time several days a week to look at questions and answer them. He felt a particular satisfaction in helping people and felt that it was a good learning opportunity to do so (P8).

Testing

The majority of participants from Biopython, Bioclipse, and Bioconductor described regularly testing their software, often as a prerequisite for contribution.

In NetLogo, we noticed more variance. Most end users who wrote NetLogo models at least relied on syntax checkers available in the NetLogo runtime environment (P14, P13, P15, P16, P11). Of that group, some performed additional testing using agent based simulation model validation techniques (P14, P13), especially when they submitted their models for publication. Participants who wrote NetLogo extensions described doing minimal testing, such as spot checking output by hand and having friends try it out to see if it worked on their computer (P11, P12). One participant explained that more extensive testing was unnecessary because it was not big enough to be worth the effort (P12).

Education and Training

Other extra work supported learning about the software. Participants described running tutorials on how to use the software to perform common analyses, mentoring other community members, and using the mailing list to learn about technologies relevant to them.

Running Tutorials

We found that participants in all four communities attended workshops, often collocated with community conferences, aimed to teach people how to use the software. Participants described running tutorials, one or two hour long training sessions where workshop participants learned how to use package APIs to work through a typical analysis on a sample dataset (P41, P38, P7, P1, P11, P27).

Mentoring Others

Several participants described mentoring new, potential members of the community (P1, P42, P7, P4, P41, P40). The most common occurrence of mentoring was GSoC, where active members of the community volunteered to mentor a student while the student proposed and developed features for the project.

The work of mentoring GSoC students typically involved corresponding with students weekly via videoconference or e-mail, over a period of three months, to discuss progress and any difficulties related to proposed goals. Mentors taught students about how to contribute to the project, as well as software engineering practices more generally. Sometimes mentoring was far-ranging, involving career choices and sample applications for jobs (P7, P1).

Studying the Mailing List

A few participants mentioned that they actively read the project mailing list to find new technical knowledge. Questions and answers about modules related to their own work served as resources that could potentially be relevant in the future (P31, P37).

Developer/User Interaction

Much extra work evolved from interactions between developers and end users. These activities included answering end users' questions, adding requested features, and suggesting fixes for others' code.

Answering Questions

Many participants described responding to users' questions related to the software. Their answers clarified how to interpret errors encountered when executing the code (P31, P37, P7), how to extend the software to provide additional functionality (P36, P44, P38, P10), or where to find a particular component that addressed the asker's particular need (P12). Participants described that they had discussions with a lot of back and forth communication, where somebody would ask for more detail, and the conversation would *"keep going like that for a while" (P36, NetLogo)*.

For paid developers of NetLogo and Bioconductor, answering user questions consumed a significant portion of their free time (P10, P42). They therefore selectively answered questions for which they were particularly knowledgeable (P27, P42, P15, P9). This strategy seemed to be a natural fit for Bioconductor package maintainers, who typically only answered questions related to their packages (P31, P37, P35) but we observed that participants from Biopython (e.g., P8, P5) used it as well. Some participants from Bioclipse made themselves available to users in real-time on Internet Relay Chat (IRC) (P29, P18).

Adding Requested Features

Participants often described receiving feature requests from end users through e-mail and over the mailing list (P5, P15, P12, P10). Requests deemed to take more time and effort needed to pass a certain threshold of priority. A request was considered low priority, for example, if there was an existing workaround to achieve the desired result (P42, P12). For instance, a participant described that he had a user request to pass matrices to his NetLogo MATLAB extension, but he didn't implement the feature because the same result could be achieved by writing a loop in NetLogo to pass multiple arrays to his extension (P12).

For projects with funding (i.e., Bioclipse, Bioconductor, NetLogo), the paid core generally considered feature requests as high priority if they aligned with objectives outlined in grants (P34, P10), or if there was a clear community need for them (P34, P10, P27, P42).

Suggesting, Making Possible Fixes to Others' Code

In some instances, participants not only reported bugs but also provided solutions. Often, participants' needs for the corrected functionality motivated their suggestions (P6, P20, P40, P3, P35, P37). Sometimes suggestions turned into pull requests:

> *"When I report issues I usually like to give as much information as possible and even point to possible fixes, and often as I go through that process I just happen to stumble-- I'm like 'Oh, this is how I would fix that' or something like that, so then a report turns into a pull request." (P20, NetLogo)*

Foreseeing User Needs

We found that participants often thought about the needs of end users. Participants modified their code so that others could reuse it, created documentation, and provided detailed snippets of source-code illustrating how to use it.

Creating Flexible Code

Participants often considered the community who would want to use their features, and the range of uses for which they might want to use it (P42, P40, P24, P2). One respondent, who described himself as *"generically selfish,"* said:

> *"…I won't code up something that I would not use. I would not code up something that only I would use and only idiosyncrasies…so I coded up things that I would use and hopefully a lot of other people would." (P24, Biopython)*

To create flexible code, participants made code syntax compatible with multiple versions of the code interpreter (P40), reused common data structures and annotation resources from other developers (P23, P4), broke the code apart into more logical pieces (P37, P11, P4), and provided access to objects via interfaces, which aren't limited to a specific implementation (P31). One participant estimated that making *"something that's maintainable and accessible"* took at least 30% more of his effort (P9, NetLogo).

Participants noted, however, that there were limitations to how much they could predict. Sometimes, making code more flexible was in response to user requests or feedback from source-code submission process (P31, P40).

Creating Documentation

Participants often documented their software contributions (P23, P15, P12). Participants from the Bioconductor and Biopython projects, which uphold certain levels of documentation quality, described creating especially extensive documentation (P35, P37, P31). Participants from Bioclipse expressed creating documentation as well, especially for setting up the Eclipse environment (P18).

We found more variability in the amount and quality of documentation in NetLogo models and extensions. Some of our NetLogo participants expressed that they kept more documentation for themselves than for others (P15, P13, P36), while others provided extensive documentation (P12, P11) and kept *readme* files up to date (P9). One reason for this variability may have been confusion about the documentation requirements (P16), as the core NetLogo team did not review contributions before accepting them, as in Biopython and Bioconductor.

Creating Examples

We found that in some cases, participants described writing additional code that demonstrated how to use their software. In Bioconductor, these examples are required for package acceptance. Contributors must write "vignettes," documents that provide a task-oriented description of the package functionality, and include executable examples.

Some NetLogo extension and model developers described creating examples as well. Extension developers provided one-line code snippets how to call functions in their code (e.g., P12). NetLogo core team members (e.g., P10) wrote whole models to illustrate to end users how to build their own. Indeed, NetLogo end users described incrementally building on these examples in order to create their own (P11, P25, P14, P16).

RQ 2: Under what conditions are scientists likely to perform (or not perform) extra work?

We found that several considerations influenced whether participants were willing to take on extra work. Participants thought about their own interests, others' interests, the relevance of the work to their jobs, their obligations, their current priorities, and their own expertise.

Does the work serve my interests?

Participants often considered whether doing extra work would benefit their own interests. They were willing to do certain kinds of extra work to publish papers describing their software. For example, maintainers of Bioconductor packages felt that *abiding by community norms* during the submission process helped produce something that was a mark of quality in the bioinformatics community (P38, P31, P22), and could be recognized as such through publication. Even though participants sensed that these so-called "software papers" would not be as well regarded as traditional publications, they felt that having a Bioconductor package imparted to them *"extra credit" (P38, Bioconductor)*. Participants who created NetLogo models described that they were willing to provide more thorough *documentation* and conduct extensive *testing* on their models submitted in conjunction with conference and journal papers if they knew that the models would also be subject to peer review (P13, P14).

Similarly, in order for current and future funders to assess their work positively, participants were willing to advertise and *promote their projects*, and *monitor code usage* (P22, P10, P34).

Participants tended to do the extra work of *suggesting fixes* when they needed working features for their own research (P20, P40, P35, P7, P3, P37). They especially escalated their efforts if a feature was high priority for them. As one participant put it:

> *"The more I need a fix, the more time and energy I'm willing to put into it." (P20, NetLogo)*

Participants seemed less likely to report or suggest fixes for issues if they were not relevant to their own research. P16, for instance, described not reporting broken models that were unrelated to his graduate thesis.

Does the work serve others' interests?

In addition to their own needs, doing extra work for free seemed to depend on whether others would benefit from it. When, for instance, developers were developing new features for themselves they often thought about whether others would need similar functionality. For example, one participant, a Bioconductor contributor, talked about an example where he asked his mentor for advice on adding a feature to his GSoC project:

> *"I sent [P42], the other author on the package, just sort of an update, 'Here, this is what I added. Do you think this is useful to the community? Would people actually use this in their workflow?'" (P44, Bioconductor)*

Other participants relied on their intuition (P41, P42, P27) or received information about others' needs in comments of their pull requests (e.g., P20, P40) and in bug reports (e.g., P20). Others *announced their work in progress* over the project mailing list in order to get feedback on whether the work would be useful to others (P5). After getting a sense how useful their contributions could be, participants *suggested fixes*, *created flexible code*, *examples* and *documentation*.

If participants considered their potential contributions too domain specific (e.g., P40, P15) or if their contribution was a personal customization (P20), they would generally not try to create a more generic solution for others.

Does the work align with my job description?
Participants who were funded to work on projects, or whose employers supported their work, tended to engage in certain kinds of extra work, such as *soliciting contributions*, *attending core team meetings*, *fixing bugs*, *mentoring others*, and *advertising and promoting the project*. From their institutes' point of view, their work on the projects served their immediate professional goals and made them look good. As the de facto leader of Biopython explained:

> *"So my employers have been quite flexible and they recognize that [Biopython] is something that is good for my career, it's something the institute is associated with in a positive way…So the practical hands-on work that's done has been related to things that we needed to do here for the research anyway." (P7, Biopython)*

If extra work instead aligned poorly with participants' jobs or career, they tended to ignore extra work such as *adding requested features* or *fixing bugs*. For example, a graduate student we spoke to mentioned that he didn't add a feature that someone else requested because he *"wasn't being paid for it." (P12, NetLogo)*

Am I under obligation?
In some cases, even when extra work was not part of participants' job descriptions, they felt an obligation to do it anyway. For example, becoming a Bioconductor package maintainer created obligations to create *examples*, conduct *testing*, *manage dependencies*, *create documentation*, and *answer questions*. Some participants found that even if they did not find extra work personally useful, peer review panels required the work as part of their criterion for publication (P13, P14). As one NetLogo modeler described:

> *"Many of the journals require if you use models in your—in the research, then they want you to submit your models, and they want them to be documented…I've never [been] a big fan of formally specifying all the units in my models. I just think it's an awful lot of work, and I hadn't found it to be all that useful, but because I have to, I do it." (P13, NetLogo)*

As we mentioned previously, social capital and reciprocity seemed to play a role in the extra work of *fixing bugs* (P29, P20). Participants seemed to feel socially obligated to conduct this extra work.

Is this extra work high enough priority?
Participants prioritized certain kinds of extra work, such as *fixing bugs*, *answering questions*, *mentoring others*, *suggesting possible fixes*, *monitoring code usage*, and *adding requested features*. Even if the work aligned well with participants' job descriptions, they had to prioritize it with respect to other tasks. P10, a core developer of NetLogo, described that critical bugs took priority over bugs with existing workarounds. NetLogo developers also described balancing user submitted bugs against requirements directly from the PI of the grant that supports NetLogo (P10, P34). Requirements originating from the PI often took priority.

Educators teaching NetLogo in the classroom (e.g., P13, P16) prioritized teaching responsibilities over extra work like *monitoring code usage* and *monitoring other people's activities*. Biopython GSoC mentors described that, in recent years, mentor time had become a limiting factor for the number of students they could accept (P7, P1). One former mentor explained that changes in his career, along with having to support his family, made mentoring GSoC students a lower priority than programming:

> *"It's a lot of time to devote; and I think people have a hard time allocating their time sometimes. You know? That's one of the reasons why I can't do as much with [mentoring] now. I have a family and I don't have the time to do that. And it's like you have a certain amount of hours for open-source stuff; and you're like 'why?' And I prefer to code I guess." (P1, Biopython)*

Do I have the expertise?
Participants explained that *suggesting possible fixes* and *answering questions* depended on whether they had the necessary expertise to do so. When participants had the expertise to contribute a fix to a project, for example, they would typically do it (P20, P40, P3). However, if participants did not have the required knowledge, they would not. A member of the NetLogo core team explained how he decided not to contribute a fix to the git project:

> *"For example, I found a bug in git once. I narrowed the bug down to a single line, fixed it for myself, and reported it, but did not contribute my fix. I'm glad I didn't too; when the bug was fixed, it required more extensive knowledge of the coding standards of git, and more extensive knowledge of the programming language (bash in this case), than I had." (P20, NetLogo)*

RQ 3: What are the design principles that can facilitate extra work?
We observed a number of common obstacles and difficulties in performing extra work. For many of them, a subset of participants had worked out effective solutions that we think can be applied more broadly. For others, existing solutions in other domains (e.g., mobile applications) seem to be a good fit to the problem. In this section, we describe a number of these problems and obstacles, design principles we think could be helpful, and the basis for this recommendation.

Use Checklists to Facilitate Learning About Community Norms and Guidelines

Common problem. In addition to learning the technical details of the scientific software under development, scientists also have to learn about the guidelines and norms of the communities. In the first section of the results we discussed that abiding by these norms and guidelines requires extra work from contributors. These guidelines varied significantly across different communities and participants often learned about these through interacting with other members or reading them on the community's websites.

We also found that there is ambiguity in interpreting these guidelines, as one NetLogo participant described:

> *"I'm not exactly sure what the standards are for [the Modeling Commons] I noticed a pretty big range of quality in the models. I definitely have downloaded a few that were literally broken and just didn't run. So I'm not sure what to make of the criteria for what makes an appropriate model to submit there." (P16, NetLogo*)

Sometimes contributors only learn about these guidelines *after* making a contribution. For example, in Biopython, contributors often learn about expected coding standards after they submit a pull request on Biopython's GitHub repository (e.g., P40). This is time consuming and only compounds the extra work that already comes with following guidelines.

Design principle. We believe many communities could benefit by borrowing an idea from Bioconductor, which maintains a short checklist of things to do before submitting a contribution. This checklist helped contributors learn and gauge the extra work required; all of our Bioconductor participants understood the expectations associated with package submission. Therefore, the extra work involved in *abiding by community norms and guidelines* may be considerably reduced by investing in checklists that help to ensure consistency and completeness in submitting contributions.

Provide Automated Support for Asking Questions

Common problem. We observed that a majority of scientists were involved in answering questions on mailing lists, which we found was another form of extra work. However, mailing lists are known to have obvious drawbacks. For instance, due to the large volume of emails, users may sometimes fail to notice certain queries of relevance to them. We found that scientists overcome some of these limitations by adopting strategies such as checking emails frequently (P42), using email filters (P31), and prioritizing questions (P34).

Design principle. Other communities may benefit from the NetLogo core team's (e.g., P34, P10) practice of using Stack Overflow, instead of mailing lists, for asking questions:

> *"People that go and ask a question on a mailing list don't usually search for the archives of the list before asking their question. Stack Overflow kind of gives you that for free because it analyzes the text of the question, automatically suggests similar questions that could have an answer that's relevant for you." (P10, NetLogo)*

Social Q&A sites such as Stack Overflow allow users to efficiently manage and answer questions. One of the biggest advantages that such Q&A sites offer is the ability to view questions that are similar and relevant to the one that is being raised. In addition, users can mention similar questions by simply specifying their URLs. Although migrating from traditional mailing lists to sites such as Stack Overflow might be one possible solution [35], it may not be entirely feasible, since scientists may not readily adopt new tools that do not directly benefit their domain task [30]. Therefore, providing this extended functionality in a way that allows users to continue using mailing lists can significantly reduce the extra work associated with *answering questions*.

Allow Contributors to Choose Between Private and Public Modes of Evaluating Submitted Contributions

Common problem. Code contributions in open-source software are often reviewed and discussed by other members before the code is accepted. The use of pull requests on GitHub to submit code contributions makes this process transparent and easier for community members to review and improve the code. However, we found that scientists are not always comfortable with making the review process public. For example, in Bioconductor, the discussions between the reviewer and the contributor happen through private emails and only the revised code is uploaded to the public repository. While this form of review takes more time than using pull requests, it allows new contributors to learn in a friendlier environment and prevents any embarrassment that they might often face (P42). However, a few Bioconductor contributors we interviewed did not mind if their code reviews were accessible to all (P31). On the other hand, a transparent review process in a community like Biopython might deter potential contributors.

Design principle. Drawing from projects with very different levels of transparency in the evaluation process, we believe that other communities may find it helpful to allow contributors to choose between these forms of evaluating code contributions rather than enforcing a protocol. In some cases, this may help facilitate *evaluating submitted contributions*.

Allow Users to Provide Feedback Within the Software, in as Few Steps Possible

Common problem. After contributions have been made, we found that feedback allows developers to evaluate how well they serve the needs of others and inform about what they should do to serve those needs. For example, P12, who developed an extension for NetLogo, mentioned that he had updated the documentation a few times after he received emails from the users. However, some participants (e.g. P9, P36) said that they rarely received feedback from users, although they were aware that their software was being used due to the number of downloads.

Design principle. We think that communities could benefit from techniques applied in the mobile applications domain, which allow users to rate and provide comments directly within the software. Examples include one-time rating popups and easy to remove banners at the top of the screen. By allowing users to effortlessly provide feedback on how they are using the software and, if possible, who they are, these designs would not only facilitate extra work around *monitoring code usage*, but also give developers more information that could help *foreseeing user needs* for other contributions. These techniques could therefore reduce the effort of users to find a channel for giving feedback to developers.

Separate Requested Features from Reported Issues

Common problem. There was a sense among our participants (e.g., P34, P35) that GitHub is a place for developers whereas the mailing list is a place for end users. In typical software development practices, bugs and requested features are reported to a central issue tracking database. In many cases, the interfaces to these databases reside on GitHub. Whereas developers may expect to find user needs in these systems, we found a portion of developers' extra work involved finding user requests on the mailing list. An additional complication with the mailing list is that developers were already conducting extra work to filter and prioritize postings of interest. If the right person does not see the message at the right time, it may never be addressed.

Design principle. It seems useful to separate requested features from reported issues in order to direct developers to requests with high user demand. As our participants pointed out, *foreseeing user needs* was an important aspect of their work, but there was a limit to what they could predict. A central dashboard dedicated to requested features may decrease the likelihood that important needs become buried under reported bugs, giving developers insight into what is needed and how many people need it. An interface, possibly a variation of the reddit (http://www.reddit.com/) format, where developers and end users can view and sort requested features by popularity, and vote on them with a simple "thumbs up" and "thumbs down" button, might prove helpful.

DISCUSSION

Based on our findings, we discuss three themes in the extra work that scientists do. The first theme is that of sharing as a continuing commitment. The second theme is that of grasping the needs of scientists and conveying developer intent. The third theme is that of efficiently distributing extra work among sharers and recipients.

Sharing as a Continuing Commitment

Some modes of sharing are an event, e.g., I give my data to a data manager, thus I have shared it and my effort is complete. Our findings illustrate that much extra work for scientific software occurs throughout the lifecycle, not only as a specific sharing event. We found that, as part of their regular work practices, when developing a new feature, participants often first considered the possible range of uses for their contributions. Consider P40, the developer who determined his code would be too domain specific to share with Biopython, P22 who wrote documentation and provided examples of his NetLogo extension in action, and P44 who sent an e-mail to the maintainer of a Bioconductor package asking if certain functionality would be useful. In many cases, the work did not stop there, particularly for Bioconductor package maintainers. Extra work actually reflected an ongoing commitment, in which participants continued to fix bugs, manage dependencies, answer questions on the mailing list, implement new features, and monitor the code usage after the software was in operation.

Some kinds of extra work seem to come at predictable intervals, whereas others do not. In Bioconductor, for instance, when preparing initial contributions, would-be contributors follow specific guidelines outlining submission requirements. *Promoting the software*, and *demonstrating its impact* are especially important when calls for funding appear. Other kinds of extra work are more dynamic, occurring in response to new uses of the software (e.g., *evaluating new submissions*, *monitoring code usage*) and changes to other tools on which the software interoperates (e.g., *managing dependencies*). We note, however, that the choices scientists make about the extra work they perform early on can affect what happens later. For example, once a scientist writing software understands a need well enough (e.g., by *announcing work in progress*), it is likely efficient at that time to *create examples*, *write documentation*, and *make the code flexible*. Otherwise, the scientist will have to spend more time later on *fixing bugs*, *adding badly needed features*, and *answering questions* that could have likely been foreseen earlier.

Grasping User Needs, Conveying Developer Intent

Knowledge about broader uses of scientific software must somehow make its way into the development process if community needs are to be well served. The participants that we interviewed considered possible future uses of their software, often relying on their own intuitions and experience (e.g., P40, P42). End users often users nudged developers by suggesting ways to make software contributions more general, reporting bugs, requesting more complete documentation, and asking for new features.

We found, however, relatively fewer instances of developers conveying the intent of their development trajectories. In many cases, developers announced new features and major releases of the software, or held workshops on analyses possible with new analysis packages—but after the software went into operation. Although developers could track activities of other developers using activity cues available in GitHub, we found that, in general, end users did not use these features. Without ways for developers to convey the intent of their development activities, potential users may simply be unaware or, or even have any way to discover, that the functionality they need exists. Broad exposure would also help increase the probability that errors are corrected before they have the opportunity to break commonly used workflows or creep into publications.

Efficiently Distributing Extra Work

It seems useful to think about efficient divisions of labor for extra work. As we mentioned in the background section, sharers are in a better position to do some extra work whereas recipients are better suited to do other extra work. Interestingly, the incentives to do the work do not necessarily line up to create an efficient division of labor, because the recipient, who, as a potential user, has a pressing current need, generally has a more obvious incentive.

Future Work

Future studies might investigate variables that influence who should do extra work. The predictability of the need for the extra work, for instance, may be of interest. If the sharer understands what is actually needed, the sharer will be in a better position to implement it, since the sharer understands the code. Another variable of interest might be how many people actually need what the extra work accomplishes. If many people will benefit, the sharer may be more inclined to do the extra work, especially if the use of the software matters to them for funding purposes (e.g., P27, P22, P29, P34). In sum, we hypothesize that if a need is predictable, and many users need it, it is more efficient for the sharer to do the work. Otherwise, the recipient should do the work because the sharer does not sufficiently understand what is actually needed, and only the recipient will benefit.

In addition, future studies might explore architectural principles for scientific software that lower the barriers to contribution and allow end user customization, thereby relaxing dependencies on the original developers. Usable APIs for instance, may make it easier for end users to contribute functionality. NetLogo end users, for example, can contribute small Java extensions without understanding the core functionality of NetLogo, which is written in Scala, another programming language. These extensions serve the contributor's needs, and potentially many others', without imposing extra work on the core developers. Toolkits [13] that allow end users to make customizations to the software but maintain the same look and feel of the core software (e.g., same user interface elements), may be valuable in instances, for example, where few users need the features.

Finally, future work might include broadening studies of extra work on software to other scientific disciplines where software sharing behaviors may be less prevalent. For instance, Velden [36] found that the field of synthetic chemistry emphasizes individual skill and personal reputation over team efforts and collective achievements. Chemical databases that specify molecular structures play a particularly crucial role in the routine work of synthesis, and discussions of important synthesis details are often kept private in order to hold competition at bay. Field specific characteristics may therefore influence questions around the conditions in which scientists are likely to perform extra work. In particular, *serving others' interests* may in fact be a disincentive, rather than an incentive, to share.

Implications for Funding Agencies

As others have argued, studies of scientific collaborative work can have important implications for funding policy [19]. This paper provides a comprehensive list of the extra work associated with developing open-source scientific software, as opposed to developing such software for oneself. The results suggest important considerations for funding scientific work of which software development is a component. The potential payoffs of extra work are substantial, provided that there exists a sizable community to use the software, because other scientists can use the software to discover new knowledge. Understanding the rich variety of extra work that scientists conduct can better inform funding policy makers of how to design practices and formulate objectives that align with these efforts. Such an understanding may also support proposal reviewers in evaluating software maintenance strategies, which are currently part of data management plans. Recognizing the value of this extra work, funders may be more willing to provide more substantial financial support to address scientific software maintenance efforts. There may also be a role for industrial funders in motivating scientists to perform extra work. Scientists interested in commercializing their software, for example, may align with industry partners who recognize its commercial potential and are willing to fund its development or pay a portion of their own developers directly to work on it. With adequate support, the software can continue effectively, building new functionality, and supporting users over time.

CONCLUSION

Our work is part of a growing body of research on the sharing and circulation of scientific resources. It is one of the few studies to examine extra effort associated with sharing software, even though software is of vital importance to nearly every scientific result. We found that both paid and unpaid contributors conducted a rich set of extra work, and that several considerations led them to do it. We also identified design principles that can facilitate extra work among sharers and recipients. In contrast to previous work showing that *reputation* for contributions to software does not matter to scientists [16,17], we found that software *use* matters to them for funding purposes. Moreover, developers might do more to convey their intent to end users if extra work is to be efficiently distributed.

Our results open up opportunities for tool design, policy, and future empirical studies of scientific software development in CSCW.

ACKNOWLEDGMENTS
We wish to thank all of our participants for their time, and our anonymous reviewers for their valuable feedback on an earlier version of this paper. We also gratefully acknowledge support of the Alfred P. Sloan Foundation, the National Science Foundation awards 1064209, 1111750, and 0943168, and the Google Open Source Programs Office.

REFERENCES

1. Altschul, S.F., Gish, W., Miller, W., Myers, E.W., and Lipman, D.J. Basic local alignment search tool. *Journal of Molecular Biology 215*, (1990), 403–410.
2. Bietz, M. Standardization in Large-Scale Collaborative Science: The Ocean Sampling Day. *Workshop on Sharing, re-use and circulation of resources in cooperative scientific work at CSCSW 2014*, (2014), 1–2.
3. Bietz, M.J., Baumer, E.P.S., and Lee, C.P. Synergizing in Cyberinfrastructure Development. *Computer Supported Cooperative Work (CSCW) 19*, 3-4 (2010), 245–281.
4. Bietz, M.J., Paine, D., and Lee, C.P. The work of developing cyberinfrastructure middleware projects. *Proceedings of the ACM 2013 conference on Computer supported cooperative work*, ACM Press (2013), 1527–1538.
5. Birnholtz, J. and Bietz, M. Data at Work: Supporting Sharing in Science and Engineering. *Proceedings of the 2003 international ACM SIGGROUP conference on Supporting group work*, (2003), 339–348.
6. Cock, P.J. a, Antao, T., Chang, J.T., et al. Biopython: freely available Python tools for computational molecular biology and bioinformatics. *Bioinformatics (Oxford, England) 25*, 11 (2009), 1422–3.
7. Corbin, J. and Strauss, J. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. Sage, 2008.
8. Crowston, K., Wei, K., Howison, J., and Wiggins, A. Free/Libre open-source software development: what we know and what we do not know. *ACM Computing Surveys 44*, 2 (2012), 1–35.
9. Dabbish, L., Stuart, C., Tsay, J., and Herbsleb, J. Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository. *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, (2012), 1277–1286.
10. Faniel, I.M. and Jacobsen, T.E. Reusing Scientific Data: How Earthquake Engineering Researchers Assess the Reusability of Colleagues' Data. *Computer Supported Cooperative Work (CSCW) 19*, 3-4 (2010), 355–375.
11. Gentleman, R.C., Carey, V.J., Bates, D.M., et al. Bioconductor: open software development for computational biology and bioinformatics. *Genome biology 5*, 10 (2004), R80.
12. Grudin, J. Why CSCSW Applications Fail: Problems in the Design and Evaluation of Organizational Interfaces. *Proceedings of the 1988 ACM conference on Computer-supported cooperative work*, (1988), 85–93.
13. Von Hippel, E. and Katz, R. Shifting Innovation to Users via Toolkits. *Management science 48*, 7 (2002), 821–833.
14. Von Hippel, E. *The Sources of Innovation*. Oxford University Press, New York, New York, USA, 1988.
15. House, N. Van, Butler, M., and Schiff, L. Cooperative Knowledge Work and Practices of Trust: Sharing Environmental Planning Data Sets. *Proceedings of the 1998 ACM conference on Computer supported cooperative work*, (1998), 335–343.
16. Howison, J. and Herbsleb, J.D. Scientific software production: incentives and collaboration. *Proceedings of the ACM 2011 conference on Computer supported cooperative work*, (2011), 513–522.
17. Howison, J. and Herbsleb, J.D. Incentives and integration in scientific software production. *Proceedings of the ACM 2013 Conference on Computer-Supported Cooperative Work*, ACM Press (2013), 459–470.
18. Huang, X., Ding, X., Lee, C.P., Lu, T., Gu, N., and Hall, S. Meanings and Boundaries of Scientific Software Sharing. *Proceedings of the ACM 2013 conference on Computer supported cooperative work*, (2013), 423–434.
19. Jackson, S.J., Steinhardt, S.B., and Buyuktur, A. Why CSCW needs science policy (and vice versa). *Proceedings of the ACM 2013 Conference on Computer-Supported Cooperative Work*, ACM Press (2013), 1113–1124.
20. Lakhani, K. and Hippel, E. Von. How open source software works: "free" user-to-user assistance. *Research policy 32*, July 2002 (2003), 923–943.
21. Lakhani, K. and Wolf, R. Why Hackers Do What They Do: Understanding Motivation and Effort in Free / Open Source Software Projects. *Perspectives on free and open source software*, (2005), 1–27.
22. Lee, C.P., Dourish, P., and Mark, G. The human infrastructure of cyberinfrastructure. *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, ACM Press (2006), 483–492.

23. Lerner, J. and Tirole, J. Some Simple Economics of Open Source. *The journal of industrial economics 50*, 2 (2002), 197–234.
24. Marlow, J., Dabbish, L., and Herbsleb, J. Impression Formation in Online Peer Production: Activity Traces and Personal Profiles in GitHub. *Proceedings of the ACM 2013 Conference on Computer-Supported Cooperative Work*, ACM Press (2013), 117–128.
25. Marlow, J. and Dabbish, L. Activity traces and signals in software developer recruitment and hiring. *Proceedings of the ACM 2013 conference on Computer supported cooperative work*, ACM Press (2013), 145–156.
26. Millerand, F., Ribes, D., Baker, K.S., and Bowker, G.C. Making an Issue out of a Standard: Storytelling Practices in a Scientific Community. *Science, Technology & Human Values 38*, 1 (2012), 7–43.
27. Orlikowski, W. Learning from notes: Organizational issues in groupware implementation. *Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, (1992), 362–369.
28. Roberts, J.A., Hann, I., Slaughter, S.A., and Donahue, J.F. Understanding the motivations, participation, and performance of open source software developers: A longitudinal study of the Apache projects. *Management science 52*, 7 (2006), 984–999.
29. Rolland, B. and Lee, C. Beyond trust and reliability: reusing data in collaborative cancer epidemiology research. *Proceedings of the ACM 2013 conference on Computer supported cooperative work*, (2013), 435–444.
30. Segal, J. Some Problems of Professional End User Developers. *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)*, Ieee (2007), 111–118.
31. Shah, S. Motivation, Governance, and the Viability of Hybrid Forms in Open Source Software Development. *Management Science 52*, 7 (2006), 1000–1014.
32. SocioCultural Research Consultants, L. Dedoose, web application for managing, analyzing, and presenting qualitative and mixed method research data. 2013.
33. Spjuth, O., Alvarsson, J., Berg, A., et al. Bioclipse 2: a scriptable integration platform for the life sciences. *BMC bioinformatics 10*, (2009), 397.
34. Spjuth, O., Helmus, T., Willighagen, E.L., et al. Bioclipse: an open source workbench for chemo- and bioinformatics. *BMC bioinformatics 8*, (2007), 59.
35. Vasilescu, B., Serebrenik, A., Devanbu, P., and Filkov, V. How social Q&A sites are changing knowledge sharing in open source software communities. *Proceedings of the 17th ACM conference on Computer supported cooperative work & social computing*, ACM Press (2014), 342–354.
36. Velden, T. Explaining field differences in openness and sharing in scientific communities. *Proceedings of the ACM 2013 Conference on Computer-Supported Cooperative Work*, ACM Press (2013), 445–458.
37. Wilensky, U. NetLogo. 1999. http://ccl.northwestern.edu/netlogo/.
38. Ye, Y. and Kishida, K. Toward an Understanding of the Motivation of Open Source Software Developers. *Proceedings of the 2003 International Conference on Software Engineering*, (2003), 419–429.
39. Zimmerman, A. Not by metadata alone: the use of diverse forms of knowledge to locate data for reuse. *International Journal on Digital Libraries 7*, 1-2 (2007), 5–16.