Leveraging Transparency

Laura Dabbish, Colleen Stuart, Jason Tsay, and James Herbsleb,
Carnegie Mellon University

// Transparency helps developers on GitHub manage their projects, handle dependencies more effectively, reduce communication needs, and figure out what requires their attention. Although transparency is not a silver bullet, it shows great promise for enhancing collaboration and coordination. //

A NEW GENERATION of development environments takes a radical approach to communication and coordination by fusing social networking functionality with flexible, distributed version control. For many years, software development environments and software architectures were designed around the idea of leaving you, the developer, in peace—free from inconsistent states resulting from colleagues' partially completed changes, free from new bugs that would complicate debugging your own code, and free from comments on and uses of your code before you deem it ready for prime time. Hiding functionality behind APIs meant you didn't have to grapple with the full complexity of code that other teams developed, nor did you have to accommodate them reaching deeply into random points in your own code.[1]

Although necessary, this isolation was never ideal for the fundamentally collaborative activity that is software development. Code hidden behind an API mostly remains a mystery to those who invoke it. Exceptional conditions that demand a more detailed understanding pose a formidable learning curve.[2] Changes that might facilitate or conflict with your own work are invisible to you until they're complete. Management practices often downplay the difficulty of integration.[3]

In response to these collaborative needs, several awareness tools and practices have arisen to support change visibility and conflict management as code evolves. Some provide signals about important activities in other workspaces,[4,5] whereas others establish norms and customs designed to keep everyone up to speed.[6] These approaches aim to strike a better balance between isolation and collaboration.

Scale and Transparency

A more radical approach is sweeping the open source world and gradually working its way into corporate environments. This new approach blends flexible version control with social media functionality to create transparent work environments, making the work visible.[7]

These transparent environments mimic social network sites, where everyone can see and have meaningful access to (almost) everything. Users can create an "interest network" in which they identify interesting people to follow and code repositories they want to watch. Events from these selected people and repositories appear in each developer's feed, keeping everyone up to date on things that interest or concern them. These environments, however, differ from the most popular social network sites in one fundamental way: the social media functionality is tightly integrated with software development tools and artifacts so that developers share code and technical artifacts, not social updates about what they had for dinner or pictures of their cats.

These transparent environments are more powerful when coupled with flexible, distributed version control systems such as Git, Mercurial, or Bazaar.

0740-7459/13/$31.00 © 2013 IEEE

TABLE 1

Cues and inferences in GitHub.

Visible cues	Social inferences	Representative quote
Recency and volume of activity	Interest and level of commitment	"This guy on Mongoid is just … a machine, he just keeps cranking out code."
Sequence of actions over time	Intention behind action	"Commits tell a story. Convey direction you are trying to go with the code … revealing what you want to do."
Attention to artifacts and people	Importance to community	"The number of people watching a project or people interested in the project obviously [means] it's a better project than something that has no one else interested in it."
Detailed information about an action	Personal relevance and impact	"If there was something [in the feed] that would preclude a feature [then] I would want it would give me a chance to add input to it."
Webpage	security.polito.it/tc/tpa	trousers.sourceforge.net

These systems do away with the idea of a single master branch, allowing developers to create many forks and pull commits from any branch into any other. Although this approach can be confusing, it lets users flexibly and selectively share and customize code. Distributed version control, combined with network graphs that show the relationships of forks, allow developers to discover interesting changes, experiment with them in separate forks, pull others' changes into their own branches, and offer changes back to the repository owner. Everyone interested in the repository can see all the branches and comment at will on repositories, commits, and issues. With sufficient effort, the tools of a previous generation can accomplish these things; the big difference is that the sharing, notification, and visibility are trivially easy.

Observing Transparency

In an effort to understand the sudden popularity of transparent environments and what makes them useful, we conducted a qualitative study of social coding among users of GitHub's free public hosting service. We interviewed and observed 24 developers managing projects on the site and split our interviews along two dimensions: hobbyists and software developers coding on open source projects as part of their job, and managers of large and small projects. (Details of our research methods appear elsewhere.[8])

In our interviews, we asked developers to walk us through their last GitHub session. Our goal was to get an idea of how they managed and contributed to projects and how they used the social functionality such as reviewing their feeds, watching projects, and following other users. We analyzed the results by looking in our data for common themes around inferences that users made, based on activity cues visible to them. Our results revealed that the transparent environment supported an effective set of coordination behaviors.

What You Can See in a Transparent Environment

Table 1 summarizes the cues and some of the primary inferences users were able to make based on those cues. For example, recency and volume of activity was generally an indicator of liveness. As with many open source hosting sites, abandoned projects greatly outnumber vibrant ones to which people continue to contribute and pay attention. Yet, developers must often compare and evaluate projects—for example, which syntax highlighting library do I choose? A search shows 158 possibilities. It can be tedious to evaluate them all, and you want to avoid depending on a dead or dying project. In GitHub, developers described getting a sense of how live or active a project was by the number of commit events in the feed. Once participant said, "Commit activity in the feeds shows that the project is alive, that people are still adding code."

Visible cues about whether someone was attending to something served as an important signal of community support. Developers interpreted activity traces of attention (following, watching, and commenting) as an indicator that the community cared about that person, project, or action. Visible information about community interest in the form of watcher and fork counts for a project was an important indicator of a project's quality and value. Several respondents indicated that they use the number of watchers or forks as a signal that a project had community interest, which helps them assess how likely it is to be good or interesting. As one developer put it, "The way you know how useful something is, is how much community there is behind it."

How Transparency Affects the Way Work Is Done

The social inferences that developers

made based on visible cues of others' behaviors supported three types of higher-level collaborative activities: project management, lessons from observation, and reputation management.

Project Management

All the developers we interviewed had GitHub projects for which they were primarily responsible. Certain types of social inferences (such as those in Table 1) supported project management activities.

Recruiting developers. Several of our respondents actively recruited others to contribute to their projects. Because forking and experimentation generally take place in public, recruitment is often fueled by information that would be invisible if this work happened in a local environment, visible only to the developer working on a private copy of the code. One user watched commits in the various forks of his project to identify skilled and committed developers. Another user, a newer GitHub member, was recruited by a project owner after submitting several good commits. The project owner began sending him tasks such as requests to address incoming issues. Intense interest in the project, inferred from a high volume of commit actions in a short period of time, sent a strong signal that a contributor was invested in the project and could be trusted to contribute more centrally. Owners grant commit rights to these interested contributors, allow new members to influence project vision, and sometimes even turn over ownership to newcomers.

Identifying user needs. Transparency also helped identify user needs by watching activity in project forks (see Figure 1). For example, one developer said he was aware that users were forking his project to fix incompatibilities with another piece of software. Based on their activity in the forks, it became clear which incompatibility issues were particularly problematic for his users: "I saw somebody trying to use [a piece of code] with Rails master. I'm like, 'Well, crap I don't know if it works with Rails master so let me check.' So that type of stuff has been useful just to get a sense of the kinds of things people might like to see." In almost all cases, these user modifications represented innovations that extended the project in interesting ways, making it compatible with other systems or more useful in general.

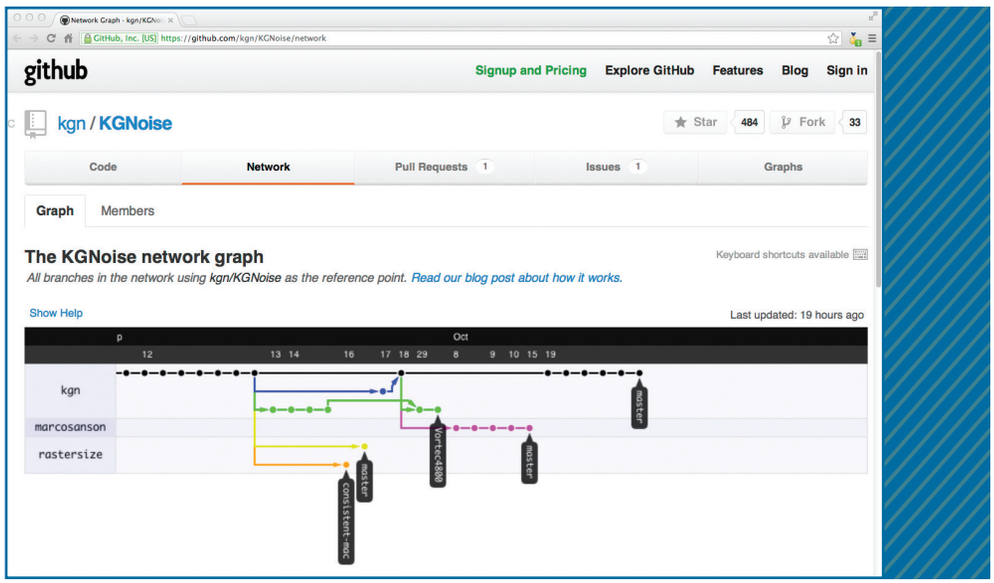

FIGURE 1. GitHub network graph of the KGNoise project (https://github.com/kgn/KGNoise). Each line represents a fork, dots are commits, and tags are labeled versions. Vertical lines represent forking and merging.

Managing incoming code contributions. Perhaps the most important project management activity developers engaged in was managing incoming code contributions. As we've already noted, users and other developers could submit changes to a project by forking the project, changing the fork, and then making a pull request (requesting that changes be merged back into the repository owner's branch). Owners were constantly making decisions about what code to accept. For very large and popular projects, owners dealt with many pull requests per day. They made inferences about the quality of a code contribution based on its style, efficiency, thoroughness (for example, was testing included?), and the submitter's track record.

Visibility across project forks took the pressure off project owners to accept all changes and allowed niche versions of a project to coexist with the official release. Thus, contributors could build directly on each other's work, even if the project owner didn't approve the changes. One developer said, "I can ignore bad changes but know that the network of experimenters can continue."

The cross-fork visibility also meant that project owners could proactively solicit changes from developers as they were working in forks and could track the status of ongoing changes. Several respondents indicated that they used the network view to identify the leading wave of changes to their project; it helped them see what people were trying to do. One participant said, "I

would look at this [network] view and actually find folks who had uploaded a patch and say, 'Hey, are you planning on sending that back to [my project]? This is what I think of it; here are some changes you could make, here are some suggestions,' and that kind of got the ball rolling."

In some cases, the changes wouldn't be submitted back because the person making the changes didn't finish doing what he or she had intended. Here, respondents said they would ping the developer to solicit a pull request or ask when the developer would finish. In some cases, if the change was incomplete but novel or useful enough, the project owner would take over and personally finish it.

In many cases, project owners needed to directly communicate about a code contribution. Sometimes, this was an attempt to solicit and motivate changes we've already described here. More often, however, this interaction consisted of negotiation around incoming pull requests. Project owners had a view of the project trajectory, and there was a need for others to buy in before making changes. Project owners would often see potential problems that a code submission could cause with other parts of the code or with changes they wanted to make in the future. In both cases, the reaction was based on implicit knowledge about code organization or on the vision for the project. One respondent said, "I could tell [one code submission] was actually going to cause some serious problems down the road, so I just responded. I always thank them because it's a big help when people contribute back; however, it wouldn't work, so I kind of explained to him why it didn't work."

The project vision and subtle code interactions were often not visible to submitters and required direct communication around the code. Similarly, the submitter's reasoning behind a change or the organization of a code submission was not always clear to the project owner. In some cases, several rounds of comments around a pull request were required to establish shared understanding of what the submitter was trying to accomplish. Fully understanding intentions and rationale was sometimes difficult through transparency alone.

> Fully understanding intentions and rationale was sometimes difficult through transparency alone.

Managing dependencies with other projects. Cross-project visibility allowed project owners to proactively manage dependencies their code had with other projects. Project owners were usually "users" of others' code, meaning that changes to those projects could affect the functioning of their own project. Accordingly, they closely watched change events from projects on which they were dependent; they watched for commit events in the feed and paid special attention to new releases and changes to files that their project used. One participant said, "[A popular website's] entire engineering team uses [my project], and so they keep an eye out for any changes as well, because when I do a release, [if] it breaks something then I essentially broke [the popular website]'s entire development for a day or something." In some cases, project owners would watch for changes they knew were coming because they had heard about them in other forums (mailing lists, blogs, and so on) or had discussed them with project owners or other developers.

When changes occurred that affected their code, developers often contacted the project owner or contributor who had made the change or joined a discussion about a proposed change For example, one project owner showed us a case where a third party chimed in on the discussion around a pull request someone else had submitted because he could tell the change affected functionality on which his company depended.

Developers would also handle conflicting or problematic changes by directly modifying the dependent project to address the problem. Transparency supported this behavior because the dependent project's code was open and accessible. The visibility of changes allowed the project owner to discover why something was no longer working. After making the change in a branch, the developer had to lobby and negotiate with the dependent project owner to get his or her changes accepted into the owner's branch.

Lessons from Observation

Transparency on GitHub allowed users to learn from other developers' actions by watching how other people coded, what others paid attention to, and how experts solved problems.

Following rock stars. Developers in our sample said they followed particular developers' actions because they deemed those developers particularly good at coding. They often referred to those developers with thousands of followers as "coding rock stars" and reported interest in how they coded and what projects they worked on. They believed their large followings often signaled exceptional skill and knowledge.

Watching watching. Developers were also interested in which projects other users were looking at; they said certain users acted as curators of the project space. As one developer put it, "I follow people if they work on interesting projects; [then] I'm interested in the projects they're interested in." Certain developers seemed to have a knack for finding useful projects in a particular interest area: "This guy has good taste in projects. … Watching him is like watching the best of objective C that GitHub has to offer."

This interest in finding the hottest new projects through what others were watching highlighted the importance that users seem to place on novelty: "I learn about new projects and new technologies way faster than ever before and it's encouraged me to get dialed in to a bunch of different tech communities I never would have had access to before."

Identifying new technical knowledge. Developers were also interested in watching other developers' actions and projects to find new technical knowledge, for example, to see how other developers had solved problems similar to theirs and how such solutions evolved: "When I find a project that solves a problem that I had and I'm going to continue to have, then I will watch it." By watching these projects and seeing the changes as they happened, users learned how their technical "neighbors" were approaching related problems, informing their own development.

Receiving direct feedback. Developers also learned from others through direct interaction: through comments on pull requests, developers got feedback about their code from more experienced developers about correctness, good form, and coding style. These interactions helped improve the code submissions' quality.

Communication also supported learning about another developer's project and getting help with attempts to build on that project. Some developers were extremely forthcoming with this type of help, checking their IRC channels and constantly issuing requests to find and address those in need. For some, this was an opportunity to grow a potential contributor, and project owners saw this as a process of ramping up users to eventually become full-fledged contributors.

Reputation Management

Actions' public visibility on GitHub led to identity management activities that centered on developers gaining greater attention and visibility for themselves and their work.

Visibility and self-promotion. Our respondents recognized the visibility of work as a valuable aspect of the community. The developers we interviewed noted the positive value of visibility, which often led to increased use of a project, extension by others, ideas from a broader audience, and exposure for the owner's other projects.

At the same time, most developers considered self-promotion (active attempts to gain additional visibility for work) as somewhat distasteful and something developers shouldn't do. Despite this collective opinion, many developers consciously managed their self-image to promote their work through consistent branding (for example, by giving their project and blog the same name or using the same Twitter handle and GitHub user ID) and by publicizing their work on platforms outside of GitHub. One user noted, "I think a lot of people that use GitHub are trying to promote themselves.… It's like, 'I have this project; you will be interested in it.'"

Some of the developers we talked to did find the attention associated with self-promotion motivating. One developer noted that watchers kept him working on something he might have otherwise abandoned: "Watching lets me know someone cares."

Being onstage. Many GitHub users have a clear awareness of the audience for their actions. This awareness influences how they behave and construct their actions—for example, making changes less frequently because they know that "everyone is watching" and could "see my changes as soon as I make them." One developer contrasted his heavily watched project with a niche project, saying that he could be more experimental with a niche project because no one was watching. Another developer directly compared it with the pressure of performing: "I try and make sure my commit messages are snappy and my code is clean because I know that a lot of people are watching.… It's like being on stage: you don't want to mess up, you're giving it your best, you've got your Hollywood smile."

Some of the developers we talked to did find the attention associated with self-promotion motivating.

Being onstage also affected how developers behaved toward other community members. Developers didn't want to offend others, for example, by publicly rejecting code contributions from long-time contributors or not following someone who followed them.

Solving Communication and Coordination Problems

Our findings suggest that transparency can make substantial inroads on three difficult communication and coordination problems in large-scale software engineering: catching potential problems early, getting a handle on high communication volumes, and knowing what needs attention.

Visibility across Micro Supply Chains

Because all artifacts are visible on the hosting site, users of a particular project can access its contents and are made continuously aware of project changes. This awareness and visibility support direct feedback and interaction between project owners and their users—what we call a micro supply chain. Visibility between the supplier (project owner) and consumer (user) means that owners can more clearly infer who their user base is, how they are using the project, and when they are having problems. Consumers are notified about changes to the product, meaning they can anticipate problematic modifications and provide immediate feedback about them. Once notified, consumers can directly communicate with the project owner about changes and discuss their consequences or request adaptations that would suit their needs. They can also directly modify the product and customize it to suit their needs with or without direct communication, if they so desire. What emerges is a highly interactive producer-consumer relationship, characterized by reciprocal dependencies. We found that transparency allows projects to evolve and become more general as a function of micro supply-chain management. Although a definitive answer awaits further research, transparency could provide significant leverage for the thorny problem of integration.

Communication when Transparency Breaks Down

If you can see something directly, and even modify and experiment with it, there's much less need for routine technical communication. When we asked developers about communication with other developers, most of these interactions seemed to occur when conflicts arose between two dependent projects or when owners and contributors were negotiating modifications to pull requests. In each case, communication seemed to happen when transparency broke down—developers needed information that they couldn't directly observe. When communication was required, they accomplished it through a variety of channels, including GitHub comments, IRC channels, Campfire, mailing lists, and so on.

Thus, although passive activity traces of others' behavior are powerful, they're limited when joint action is required. In part, this is owing to the lack of feedback or interactivity these visible traces provide. Our results suggest these traces support rich inferences about individuals and repositories. However, when new collaborative actions involve dependencies, two-way communication is required.

> Transparency could provide significant leverage for the thorny problem of integration.

Signals of Attention

Visible signals of attention provided notification of other developers' behavior and seemed to help users manage the downsides of transparency across a large-scale network. They helped developers identify projects and events they found interesting or useful. These signals, when aggregated, also gave some users higher levels of status because they indicated community approval or admiration. As one user put it, by visibly watching a repository, "I'm kind of giving them some token of my attention. I'm saying, 'I like what you're doing.'" Signals of attention functioned to provide awareness of what other users cared about or were looking at.[9]

As powerful and useful as transparency seems to be, it's certainly not a cure for all ills. In our study, developers still reported problems with information overload, especially if they watched several very active repositories or followed many active people. Feeds with updates about interest networks are powerful, but these too can be swamped when projects grow large enough or a developer wants to monitor many projects and users.

It's also clear from our interviews that not everyone is comfortable living onstage all the time. Moreover, the level of discomfort is almost certainly underrepresented in our sample, because all of our interviewees had voluntarily moved to a transparent environment.

Based on our observations, developers and development managers can take several steps to fully leverage transparent environments:

- Companies can recommend particular developers to follow as exemplars of sound practice and style.
- New developers can receive a planned program of "asynchronous

mentoring," exposing them to critical skills through recommendations of repositories to watch and people to follow.
- Assigned mentors can follow new developers, providing a lightweight way to advise and encourage.
- Developers must take extra care to explicitly document the rationale for a change and vision for a project, which aren't always readily apparent.
- Development organizations can coordinate watching and following relations with product road-mapping activities so that evolving dependencies are carefully attended to.

Modularity and information hiding—revolutionary ideas in their time—remain, decades later, among the most significant conceptual tools we have for coordinating development work. Although the evidence isn't yet all in, it seems to us that the complementary idea of transparency could rank among the breakthroughs of our day.

Acknowledgments

We gratefully acknowledge support from NSF grants IIS-1111750, SMA-1064209, OCI-0943168, and CNS-1040801 and grants to Dabbish and Herbsleb from the Center for the Future of Work, Heinz College, Carnegie Mellon University.

References

1. D.L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Comm. ACM*, vol. 15, no. 12, 1972, pp. 1053-1058.
2. C.R.N. de Souza et al., "Sometimes You Need to See through Walls: A Field Study of Application Programming Interfaces," *Proc. ACM Conf. Computer-Supported Cooperative Work*, ACM, 2004, pp. 63–72.
3. R.E. Grinter, "Recomposition: Putting It All Back Together Again," *Proc. ACM Conf. Computer Supported Cooperative Work*, ACM, 1998, pp. 393–402.
4. A. Sarma, Z. Noroozi, and A. van der Hoek, "Palantír: Raising Awareness among Configuration Management Workspaces," *Proc. Int'l Conf. Software Eng.*, IEEE CS, 2003, pp. 444–454.
5. Y. Brun et al., "Proactive Detection of Collaboration Conflicts," *Proc. ACM Conf. Foundations of Software Eng.*, ACM, 2011, pp. 168–178.
6. C. Gutwin, R. Penner, and K. Schneider, "Group Awareness in Distributed Software Development," *Proc. ACM Conf. Computer-Supported Cooperative Work*, ACM, 2004, pp. 72–81.
7. A. Parker, S.P. Borgatti, and R. Cross, "Making Invisible Work Visible: Using Social Network Analysis to Support Strategic Collaboration," *California Management Rev.*, vol. 44, no. 2, 2002, pp. 25–46.
8. L. Dabbish et al., "Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository," *Proc. ACM Conf. Computer-Supported Cooperative Work*, ACM, 2012, pp. 1277–1286.
9. T. Erickson and W.A. Kellogg, "Social Translucence: An Approach to Designing Systems that Support Social Processes," *ACM Trans. Computer-Human Interaction*, vol. 7, no. 1, 2000, pp. 59–83.

ABOUT THE AUTHORS

LAURA DABBISH is an assistant professor of Information Technology and Organizations at the H. John Heinz III College and the Human-Computer Interaction Institute in the School of Computer Science at Carnegie Mellon University. Her research interests include new forms of technology-mediated organizations, social technology design, and awareness and coordination in knowledge-intensive work. Dabbish received a PhD in human-computer interaction from Carnegie Mellon University. Contact her at dabbish@cmu.edu

COLLEEN STUART is a postdoctoral Fellow in the Human-Computer Interaction Institute at Carnegie Mellon University. Her research interests include how social relationships and digital transparency influences collaborative work. Stuart received a PhD in organizational behavior from the University of Toronto. Contact her at hcstuart@cs.cmu.edu.

JASON TSAY is a PhD candidate in software engineering with the Institute for Software Research at Carnegie Mellon University. His research interests include software–developer collaboration and development environments. Tsay received a BS in computer engineering from the University of Texas at Austin. Contact him at jtsay@cs.cmu.edu.

JAMES HERBSLEB is a professor in the Institute for Software Research, School of Computer Science, and the Center for the Future of Work, Heinz College, Carnegie Mellon University. His research interests include collaboration and coordination in large-scale software engineering, particularly in the context of sociotechnical ecosystems. Herbsleb received a PhD in psychology and a JD in law from the University of Nebraska. He's a member of ACM. Contact him at jdh@cs.cmu.edu.

See www.computer.org/software-multimedia for multimedia content related to this article.