specifically keen to learn if there were any quirks in the process that we could ease. If we could flatten the process out a little, we could make it easier for the community to participate.

To do this, we sat down and watched a contributor working with bugs. We noted how he interacted with the bug tracker, what content he added, where he made mistakes, and other elements. This data gave us a solid idea of areas of redundancy in how he interacted with a community facility.

What Jorge on my team did here was user-based testing, more commonly known as *usability testing*. This is a user-centered design method that helps evaluate software by having real people use it and provide feedback. By simply sitting a few people in front of your software and having them try it out, usability testing can provide valuable feedback for a design before too much is invested in coding a bad solution.

Usability testing is important for two reasons. The most obvious is that it gets us feedback from a lot of real users, all doing the same thing. Even though we aren't necessarily looking for statistical significance, recognizing usage patterns can help the designer or developer begin thinking about how to solve the problem in a more usable way.

The second reason is that usability testing, when done early in the development cycle, can save a lot of community resources. Catching usability problems in the design phase can save development time normally lost to rewriting a bad component. Catching usability problems early in a release cycle can preempt bug submissions and save time triaging. This is on top of the added benefit that many users may never experience such usability issues, because they are caught and fixed so early.

Open source is a naturally user-centered community. We rely on user feedback to help test software and influence future development directions. A weakness of traditional usability testing is that it takes a lot of time to plan and conduct a formal laboratory test. With the highly iterative and aggressive release cycles some open source projects follow, it is sometimes difficult to provide a timely report on usability testing results. Some examples of projects that overcame problems in timing and cost appear in the accompanying sidebar ("Examples of Low-Budget, Rigorous Usability Tests") by Celeste Lyn Paul, a senior interaction architect at User-Centered Design, Inc. She helps make software easier to use by understanding the user's work processes and designing interactive systems to fit their needs. She is also involved in open source software and leads the KDE Usability Project, mentors for the OpenUsability Season of Usability program, and serves on the Kubuntu Council.

EXAMPLES OF LOW-BUDGET, RIGOROUS USABILITY TESTS

There are some ways you can make usability testing work in the open source community. Throughout my career in open source, I have run a number of usability tests, and not all have been the conventional laboratory-based testing you often think of when you hear "usability test." These three

examples help describe the different ways usability testing can be conducted and how it can fit into the open source community.

My first example is the usability testing of the Kubuntu version of Ubiquity, the Ubuntu installer. This usability test was organized as a graduate class activity at the University of Baltimore. I worked with the students to design a research plan, recruit participants, run the test, and analyze the results. Finally, all of the project reports were collated into a single report, which was presented to the Ubuntu community. The timing of the test was aligned with a recent release and development summit, and so even though the logistics of the usability test spanned several weeks, the results provided to the Ubuntu community were timely and relevant.

Although this is the more rare case of how to organize open source usability testing, involving university students in open source usability testing provides three key benefits. The open source project benefits from a more formal usability test, which is otherwise difficult to obtain; the university students get experience testing a real product, which looks good on a curriculum vitae; and the university students get exposure to open source, which could potentially lead to interest in further contribution in the future.

My second example involves guerilla-style usability testing over IRC. I was working with Konstantinos Smanis on the design and development of KGRUBEditor. Unlike most software, which usually are in the maintenance phase, we had the opportunity to design the application from scratch. While we were designing certain interactive components, we were unsure which of the two design options was the most intuitive. Konstantinos coded and packaged dummy prototypes of the two interactive methods while I recruited and interviewed several people on IRC, guiding them through the test scenario and recording their actions and feedback. The results we gathered from the impromptu testing helped us make a decision about which design to use.

The IRC testing provided a quick and dirty way of testing interface design ideas in an interactive prototype. However, this method was limited in the type of testing we could do and the amount of feedback we could collect. Remote usability testing provides the benefit of anytime, anywhere, anyone at the cost of high-bandwidth communication with the participant and control over the testing environment.

My final example is the case of usability testing with the DC Ubuntu Local Community (LoCo). I developed a short usability testing plan that had participants complete a small task that would take approximately 15 minutes to complete. LoCo members brought a friend or family member to the LoCo's Ubuntu lab at a local library. Before the testing sessions, I worked with the LoCo members and gave them some tips on how to take their guest through the test scenario. Then, each LoCo member led their guest through the scenario while I took notes about what the participant said and did. Afterward, the LoCo members discussed what they saw in testing, and with assistance, came up with a few key problems they found in the software.

The LoCo-based usability test was a great way to involve nontechnical members of the Ubuntu community and provide them an avenue to directly contribute. The drawback to this method is that it takes a lot of planning and coordination: I had to develop a testing plan that was short but provided

enough task to get useful data, find a place to test (we were lucky enough to already have an Ubuntu lab), and get enough LoCo members involved to make testing worthwhile.

—Celeste Lyn Paul
Senior Interaction Architect
User-Centered Design, Inc.

Although Celeste was largely testing end-user software, the approach that she took was very community-focused. The heart of her approach involved community collaboration, not only to highlight problems in the interface but also to identify better ways of approaching the same task.

These same tests should be made against your own community facilities. Consider some of the following topics for these kinds of observational tests:

- Ask a member to find something on your website.
- Ask a prospective contributor to join the community and find the resources they need.
- Ask a member to find a piece of information, such as a bug, message on a mailing list, or another resource.
- Ask a member to escalate an issue to a governance council.

Each of these different tasks will be interpreted and executed in different ways. By sitting down and watching your community performing these tasks, you will invariably find areas of improvement.

Measuring Mechanics

The lifeblood of communities, and particularly collaborative ones, is communication. It is the flow of conversation that builds healthy communities, but these conversations can and do stretch well beyond mere words and sentences. All communities have collaborative mechanics that define how people do things together. An example of this in software development communities is bugs. Bugs are the defects, problems, and other it-really-shouldn't-work-that-way annoyances that tend to infiltrate the software development process.

Every mechanic (method of collaborating) in your community is like a conveyor belt. There is a set of steps and elements that comprise the conversation. When we understand these steps in the conversation, we can often identify hooks that we can use to get data. With this data we can then make improvements to optimize the flow of conversation.

Let's look at our example of bugs to illustrate this.

Every bug has a lifeline, and that lifeline is broadly divided into three areas: *reporting*, *triaging*, and *fixing*. Each of these three areas has a series of steps involved. Let's look at reporting as an example. These are the steps: