EDITOR DIRK RIEHLE
Friedrich Alexander-University of Erlangen Nürnberg;
dirk.riehle@fau.de

Open Source Software Engineering the Eclipse Way

Wayne Beaton, The Eclipse Foundation

In open source software engineering, the source code that is made publicly available is developed using processes that actively engage and solicit participation by a community.

There is a perception in some circles that software engineering in open source is wild, eclectic, and undisciplined. In many cases, the proverbial software developers hacking in their garage is real. The fact is, however, that there exists a great variety of practices in open source software engineering with varying degrees of rigor and maturity in engineering practices.

Hallmark practices of traditional software engineering, including build automation, continuous integration, reproducible builds, integration and regression testing, and performance testing, are leveraged in open source. Open source projects of various sizes, compositions, and industrial investments have varying degrees of rigor. This is especially true in the modern environment of service providers who make their services available without fees for open source software projects. For example, even the smallest open source software project can afford to host its content on commercial-grade source code management systems and issue trackers, leverage commercial-grade continuous integration systems, and ship builds via commercial-grade software repositories.

But open source software engineering is more than leveraging a handful of royalty-free support technologies. The Open Source Initiative defines open source in terms of the consumer. That is, the focus of their definition is the rights granted to the recipients of the source code; specifically, software is considered to be open source when the source code is publicly distributed under a royalty-free license that allows its use, study, modification, and redistribution. Under this definition, an entirely closed software development team that works in private, using proprietary development methodology and leveraging traditional software development engineering practices, can be said to produce open source software so long as the source code produced is royalty free, under the terms of

Digital Object Identifier 10.1109/MC.2021.3069530
Date of current version: 4 June 2021

FROM THE EDITOR

Welcome back to "Open Source Expanded" and the current theme of open source communities! After Jesus Gonzalez-Barahona has reviewed the history of community open source, and Isabel Drost-Fromm and colleagues have discussed governance at The Apache Software Foundation, it is now my honor to have Wayne Beaton explain how open source software development works at the Eclipse Foundation. He dives into engineering best practices, providing insights from one of the most successful open source foundations of today. Next up will be community health metrics. Happy hacking, everyone, and please stay healthy! *— Dirk Riehle*

an open source software license. This is colloquially referred to as "throwing it over the wall."

OPEN ENGINEERING

Open source software engineering is the practice of creating open source software using an open process. An open process facilitates some degree of open collaboration between developers representing a diversity of thought and motivation. That is, in open source software engineering, the source code of the software that is made publicly available under an open source license is developed in a forge that is also publicly accessible, using processes that actively engage and solicit participation by a community.

Many, perhaps most, open source projects describe themselves as "agile" by some definition of the term. That may mean that they work according to the Agile Manifesto[1] or employ some formal or informal agile development methodology. However it is defined, agile development isn't specific to open source; indeed, agile development methodologies can be employed in a fully private software development context. So, while "agile" is certainly a common characteristic of many open source projects, it isn't a defining one.

When speaking of open source software engineering, the notion of community frequently comes up. This nebulous term *community* is generally understood to mean the individuals and organizations that form around an open source project. The community is the users and organizations that adopt open source software and incorporate it into their own projects or products. The community is the primary source of contribution to an open source project, including both the developers who contribute patches and other content to the project and the project team themselves.

Open source software engineering is largely defined by engagement with the community. The degree to which an open source project team engages with a community varies according to the goals of the project.

There are several variables that an open source project can tune to manage its level of community engagement and inclusivity:

- transparency
- openness
- eliminating barriers
- shipping software.

TRANSPARENCY

Transparency is the practice of showing the community what the team is doing. The general idea is to give anybody who cares to pay attention to the project the opportunity to understand the project team's work.

The act of making source code publicly available under the terms of a recognized open source license certainly meets the definition of transparency. However, making the decision-making processes behind the development of that software publicly available meets the definition even more. Exposing the decision-making process adds predictability for the community, which makes it easier for adopters to set their own development timelines and for the project team to build user excitement for their releases.

Open source projects use a variety of means of open communication to give their community of users and adopters an opportunity to monitor their decision-making process. It's common for open source project teams to host calls and then capture the minutes from their discussion in one of their public channels. With the emergence of open source foundations and foundries-as-a-service, expectations for transparency have evolved to include publicly accessible issue trackers. However, the true measure of transparency is the degree to which these public-facing tools are used to accurately disclose the decision-making processes of the project.

OPENNESS

Openness is the practice of letting others participate. That is, "open" in this context means "open to all comers." Like all practices in open source, there is a range or degree of openness.

Perhaps the most important thing that an open source project can do to make it possible for members of its community to participate is to use the same source code repository as the community. When developers push their content directly (and frequently) into a repository that the rest of the community has access to, so that the entire community has access to the most up-to-date version of the project content, members of that community have the ability to keep up with project development and an equal opportunity to contribute. Put another way, when a project team works in private and then periodically synchronizes its internal repository with a public one, it is basically impossible for anybody outside of the private team to contribute any content.

Making an open source project's repositories publicly available and accessible does not necessarily mean that everybody has equal privileges. In an open-collaboration scenario, it is possible for members of the community to earn additional privileges by demonstrating knowledge of the open source project's code base and development practices. For example, community members may make contributions to an open source project, but those contributions need to be received, reviewed, and accepted by a member of the project's development team. Over time, a contributor who has demonstrated knowledge of the project's code base and an understanding of the project's rules of engagement would be invited to join the project team.

Likewise, an issue tracker that is publicly available and accessible to the community makes it possible for the entire community to participate on equal footing. Open-issue tracking lets the community raise issues, provide direct feedback, and participate in planning. In essence, keeping source repositories, issue trackers, and communication channels open for community participation makes the project team a part of the community and not separate from it.

ELIMINATING BARRIERS

To be open, an open source project should be careful to lower barriers for participation. Barriers for participation may or may not be obvious. Every service that an open source project team leverages likely has some terms of use that may be a barrier for some. That's not to say that an open source project should not leverage available services but that a project team should be cognizant of what segments of the community they might be excluding with their choices.

Open source projects that have a goal of widespread adoption need to pay attention to intellectual property management. Source code is a form of intellectual property. As with all forms of intellectual property, source code must be licensed. Care must be taken to select the license best suited for the project's goals. The software leveraged by an open source project, the so-called "third-party software," is itself licensed. The adoption of third-party software with licenses that conflict with the project license or with the licenses of other third-party software puts adopters at potential legal risk. A license might, for example, allow the use of a bit of intellectual property under a certain set of circumstances but not others. Or it may place requirements on consumers or make specific requirements of derivative works or linked code. These, again, are potential barriers.

Exposing the decision-making process adds predictability for the community, which makes it easier for adopters to set their own development timelines and for the project team to build user excitement for their releases.

A barrier for entry that may not be obvious is vendor domination, both real and perceived. It may be difficult, for example, for some potential contributors to even adopt open source software that is controlled by their competitors. Operating in a vendor-neutral manner is an important factor in eliminating barriers for entry. This is one of the ways in which open source foundations play an important role. Open source foundations are a means of disconnecting an open source project from direct association with one specific vendor and opening it up to more general participation.

SHIP SOFTWARE

A core value of open source software engineering is engagement with the community as a source of feedback and facilitation of participation. Shipping software is the means of engaging with the broadest possible cross-section of the community.

With every single commit to a publicly accessible Git repository being immediately accessible to the community, coupled with continuous integration build infrastructure, an open source project operating in an open manner can be thought of as always shipping. For some open source projects, this may be enough.

Setting up formal releases introduces greater predictability and provides opportunity to improve quality for users and adopters. Formal nightly and integration builds give adopters an opportunity to test early and frequently; interim milestone builds provide an opportunity to engage in more thorough testing, give the project team a means of practicing their release processes, and, with the increased quality over nightly and integration builds, provide an opportunity to engage with a larger part of community.

The means by which an open source project ships software varies just like the degree to which a project operates in an open and transparent manner and eliminates barriers varies based on the nature of the technology and how broadly the project wants to engage with its community.

THE ECLIPSE WAY

What we know today as the Eclipse integrated development environment has roots that go back to 1998 in closed source development. In 2001, the software was released into open source. While innovating a new platform and growing an ecosystem, the original Eclipse project team also innovated a new method for developing software in open source. Many of the practices in what came to be known as "The Eclipse Way"[2] found their roots in extreme programming[3] and principles that became the Agile Manifesto. The Eclipse project has shipped high-quality open source software on time for 20 years and counting.

To drive the success of its open source software, the development team drove the creation of an ecosystem of users, extenders, and adopters—individuals and organizations building their own products based on the platform. That ecosystem and community effectively became the customer.

What is the secret of their success? While they share numerous practices that are common across open source projects, the Eclipse project team highlights five key practices that contribute to their success:

- milestone builds
- planning
- continuous testing
- endgame
- decompression.

MILESTONE BUILDS

Shipping code is a defining factor of the Eclipse Way. Shipping code means getting the code in front of the consumer frequently, soliciting feedback, and integrating that feedback quickly and frequently. To that end, the Eclipse project adopted a practice of producing regular milestone builds on its path to producing a final annual release. In the very early days, milestone builds were produced every four weeks, but experience led the team to expand this to six weeks with a decision to adopt a cadence of quarterly releases, and the time between each milestone build shrunk to three weeks.

Each milestone is the culmination of a development cycle that is treated as a miniature release with distinct planning, development, and stabilization phases. The end product of every six-week development cycle is a high-quality milestone build that is good enough to be used by the community.

When the Eclipse platform released annually, the schedule started with a few weeks of planning, followed by seven milestones, and concluded with the end game (later). With the completion of milestone six came an application programming interface (API) freeze, and with milestone seven a feature freeze. With the switch to quarterly releases, the API and feature freezes now come with the third milestone. Following feature freeze, no new work is performed, and all of the developers focus their attention on polishing their work, stabilizing features, and performance testing. By delivering regular milestones and having a well-defined feature and API freeze milestone, the Eclipse project balances agility with predictability so that adopters have stability in their own plans.

PLANNING

With the Eclipse Way, all planning is done in the open, and there are no private channels. The plans themselves are publicly accessible. The Eclipse project is actually a collection of projects that works together. There is a separate project, for example, focused on creating the Eclipse platform, a project focused on creating the Eclipse development tools for Java, and a project focused on creating the Eclipse plug-in development tools. With the addition of the Eclipse open source projects that participate in the simultaneous release, this extends to include almost 100 distinct open source projects, all collaborating together to produce high-quality software on time. Each of these projects has its own developers and project leads.

Early planning focuses on broad themes. Themes may be general, like improve the user interface or specific, like support Java 15. Everybody, including the committers, project leadership, and members of the community, works together to determine the themes, and, from that, individual project teams work out their own project plans. Project plans feed into milestone plans, with specific plan items being assigned to each milestone.

Planning is incremental and iterative. The project teams ultimately decide what work will be done, but decisions are made within the framework of the overall plan, strategic goals of their stakeholders, and input from users, adopters, and the rest of the community.

Plans are dynamic. The Eclipse project values high-quality and on-time delivery above all else, so sometimes plan items are changed, deferred, or dropped from milestones and the release. Individual project plans are updated as development progresses and communicated to the community. Everything happens in the open. Plans are only ever marked as "final" at the end of the release.

CONTINUOUS TESTING

With high quality being a critical value of the Eclipse Way, continuous testing is required. From the very beginning, the Eclipse project produced fully automated builds in various flavors, each with its own specific purpose.

Build and test failures are expected in nightly builds; nightly builds are not generally intended to be consumed by anybody but, rather, to provide important feedback directly to the project team. The intention is to resolve problems early while they're still small.

In weekly integration builds, features are in a state where the whole development team can use and work with them. Expectations are that all automated tests will be successful and that the project team will use the results from each integration build to build the next one.

Milestone builds are produced at the end of the development cycle. They are expected to be of sufficient quality for the community to use; they are intended to solicit feedback from the community that can be integrated into the planning for subsequent development cycles.

With the Eclipse Way, the product is always in beta. That is, the product itself may not be of the high quality expected in a milestone or a release, but it always works. In many ways, every commit can be thought of as shipped code. Developers who wish to work on the very bleeding edge can pull the code at any point in the development cycle, build it, and have a reasonably high probability of success.

Having a solid set of unit tests allows the project team to innovate and refactor code with confidence. Unit testing is a critical aspect. The Eclipse platform has more than 100,000 JUnit tests divided across the many projects, repositories, and components to test the correctness of behavior. Developers regularly run subsets of tests on their own workstations. The full suite of tests is run after every build; integration and milestone builds are only considered complete when all tests pass.

In addition to unit tests, resource tests are run to mitigate the risks associated with memory leaks and overconsumption of resources, and API verification tests ensure that APIs remain stable and that cases where components break API boundaries and illegal use internal/non API code are identified. Unit, resource, and API verification tests are run with every nightly, integration, and milestone build.

Performance tests are run to identify performance regressions; a database of performance results is maintained from build to build to ensure that regressions are identified quickly. Performance tests are expensive and so are only run for weekly integration and milestone builds.

To encourage the adoption of milestone builds, the developers publish a new and noteworthy document that advertises new features to motivate both users and adopters to take their important roles in the feedback loop. A new and noteworthy document is produced with each milestone build; the milestone new and noteworthy documents are combined into aggregate new and noteworthy for the release.

Working with the community is a recurring theme. In some cases, an open source project can be successful with a "build it and they will come" sort of attitude, but getting real community feedback requires sustained investment from the developers themselves. One of the important lessons of the Eclipse Way is that the developers are the best evangelists.

ENDGAME

After the last milestone build, the Eclipse Way enters the endgame. After the last milestone, the developers stop doing development and switch into a mode of rigorous testing and mitigation. During this phase, the project team will produce between three and four release candidate builds of increasing quality. Like every other build, all release candidates are available to the community, and the community is invited to participate and provide feedback. The final release candidate becomes the generally available release.

No new functionality is created during the endgame. In between testing, the developers focus on improving the documentation and help and on building the aggregated new and noteworthy document.

DECOMPRESSION

Following the release, when following the Eclipse Way, the team enters the decompression phase. In the early days of a single annual release in June, the decompression phase coincided with the arrival of summer in the northern hemisphere and, accordingly, summer vacation. After an exciting and fulfilling year of discovery and creation, the team would take time to recover for the sanity of the developers. Time to breathe.

During this time, the team engages in a retrospective of the last cycle, documenting its achievements and failures, reviewing its process, and looking for opportunities to improve cross-team collaboration. The retrospective itself is a publicly accessible document (for example, The Eclipse Helios Retrospective). The process itself is continually reevaluated and evolved. The development team has influence on how the process evolves (the Eclipse project's committers decided, for example, to switch to quarterly releases). Note that, in practice, the individual project teams tend to engage in a formal retrospective every few releases. It is also during the decompression period that the team starts to work with the community to assemble the plan for the next release.

The Eclipse project's practices have evolved over time. Six-week development cycles producing annual releases gave way to three-week development cycles producing quarterly releases. The project team has shared more control over the years to the point where the various project teams that collaborate in the Eclipse project represent the interests of multiple organizations all working together.

There are many ways to create open source software. The choice of development methodology is certainly an important consideration but is not a defining characteristic of open source software engineering. Rather, open source software engineering is a practice of working in an open and transparent manner and inviting a community to participate in some manner.

The manner or degree to which an open source project engages its community varies. Operating in an open and transparent manner, lowering barriers and inviting contribution, and sharing control are great ways to grow a community, but they introduce challenges. Open source project teams need to have goals with respect to community engagement and set (and evolve) their practices accordingly. C

REFERENCES

1. Manifesto for Agile Software Development. Accessed: Mar. 1, 2021. [Online]. Available: https://agilemanifesto.org/
2. D. Megert, *The Eclipse Way*. Eclipse Summit India. 2016. https://codeandtalk.com/v/eclipsesummit-india-2016/the-eclipse-way-by-daniel-megert-at-eclipsesummit (accessed Mar. 1, 2021).
3. K. Beck, *Extreme Programming Explained: Embrace Change*. Reading, MA: Addison-Wesley Professional, 1999.

WAYNE BEATON is the director of Open Source Projects, The Eclipse Foundation, Ottawa, Ontario, K2H 1B2, Canada. Contact him at wayne.beaton@eclipse-foundation.org.