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

How to Select Open Source Components

Diomidis Spinellis, Athens University of Economics and Business

With millions of open source projects available on forges such as GitHub, it may be difficult to select those that best match your requirements. Examining each project's product and development process can help you confidently select the open source projects required for your work.

If most of the code comprising your product or service isn't open source software, it's highly likely that you're wasting effort and cash reinventing the wheel. Yet with millions of open source projects available on forges such as GitHub, it may be difficult to select those that best match your requirements. Examining two facets of each candidate project, the product and its development process (see Table 1), can help you select with confidence the open source projects required for your work.

PRODUCT

Functionality

Begin by assessing the functionality of the project under consideration and determine whether it covers both current needs and future strategic directions. For instance, if you are selecting a message queue, consider whether the underlying messaging protocol matches the one prevalent in your industry and whether the system can scale in the future to cover your projected needs. It is equally important to evaluate whether the project's functionality is egregiously excessive compared to your needs. For example, if you simply want to compress data that you store in a file, you may not want to use a multiformat data archiving library. Selecting a small, focused project over a larger one has many advantages. In typical cases, such a choice will offer a reduced storage footprint for your system, fewer transitive third-party dependencies, a lower installation complexity, and a smaller surface vulnerable to malicious attacks.

If an open source project's functionality nearly fits your organization's needs and no other project can satisfy them completely, you can still use it and make the required changes on your own. However, under this scenario, you must more stringently evaluate the elements I outline later on regarding source code changes and contributions. See the last column in Table 1.

Digital Object Identifier 10.1109/MC.2019.2940809
Date of current version: 22 November 2019

FROM THE EDITOR

Welcome back! Open source gives you high-quality software for free. What's not to like about this? But wait a second: You need to choose the right open source component. Making a poor choice for using an open source component in your products or projects can create serious problems. In this article, well-known open source expert Diomidis Spinellis takes us through the process of selecting the right open source component for your needs. Significant thought should be spent on such a decision, because using an open source component creates a dependency that needs to be managed, and some dependencies are easier to manage than others. But more on this topic in one of the next columns. As always, happy hacking! — *Dirk Riehle*

Licensing

Narrow down your search by examining whether the project's licensing[1] is compatible with your business model, mission, or other software you are using. Within your project's source code, if you directly incorporate elements licensed under the GNU General Public License, then you also must distribute your code under the same license. This may be undesirable if your business model depends on keeping your product's source code under wraps; in this case, you should be looking for projects that use more permissive licenses, such as the Berkeley Software Distribution and Apache ones. Similar concerns apply if you are offering software as a service, and you plan to use software licensed under the Affero General Public License. As another example, software released under version 1.1 of the Mozilla Public License cannot be linked together with code licensed under the GNU General Public License.

Nonfunctional properties

Evaluate the project's fit with your requirements by also looking at its nonfunctional properties. Is it compatible with your product's processor architecture, operating system, and middleware? Will it accommodate your future expansion plans and directions? For example, if your product works on macOS but you're also eyeing the Windows market, then you should be using open source libraries supported on both systems. Is the product's performance compatible with your requirements? This is especially important when selecting a database or a big data analytics infrastructure. If performance is critical, do not assume particular performance outcomes; rather, benchmark with realistic workloads.

Popularity

Then consider the project's popularity. Popularity is important because it can determine how likely it will be for your questions to receive answers on public forums, for volunteers to contribute fixes and enhancements, and for the project to continue to evolve if its original developers veer off course (namely, losing interest or steering the project toward an undesirable direction). Simple metrics, such as GitHub stars, the number of StackOverflow questions with the corresponding tag, the download count, and the number of Google query results are all usually sufficient to discern the cases that really matter.

Documentation

The project's documentation is another aspect that should be examined. Although most answers regarding a software's operation ultimately lie in the source code, resorting to such digging for everyday operations is undesirable. Therefore, judge how well the software is documented, both at the technical (installation and maintenance procedures) and user levels (tutorials and reference manuals). Although nearly all mature open source software projects are well documented, some smaller ones suffer in this dimension. There are Unix command-line utilities, for example, that lack the traditional manual page. I try to avoid such projects, both to keep my sanity (life is too short to waste on hunting down command-line options) and because such a level of indifference toward the end user is often a sign of deeper problems.

Source code

This brings me to another product characteristic you should check, namely the project's source code and

TABLE 1. Judging the open source project selection criteria.

	Attribute	Deal breaker	Areas that may require investment	Areas important for in-house development
Product	Functionality	Partial	🟨	
	Licensing	Full		
	Nonfunctional properties	Partial		
	Popularity		🟨	
	Documentation		🟨	
	Code quality		🟨	🟩
	Build system			🟩
Process	Development process			
	Code commits			
	Project releases			
	Support			
	Issue management		🟨	🟩
	Acceptance of contributions			🟩

the code's quality. If you anticipate adjusting the project to your needs, then select projects written in programming languages with which you are familiar. Even if you don't plan to touch the project's source code, low code quality can affect you through bugs, security vulnerabilities, poor performance, and maintenance problems. Again, there's no need to dig deeply to form a useful opinion. In most cases, your objective is to avoid problematic projects, not to perform thorough due diligence of the code. Look at the project's source code files. Are they named and organized into directories following the conventions of the project's programming language? Is there evidence of unit testing? Does the repository also contain elements that it shouldn't, such as object and executable files? Open and browse a few files. Are methods or functions short and readable? Are identifiers well chosen? Is the code reasonably commented? Is the formatting consistent with the language's coding conventions? Again, serious deviations are often indicators of more important hidden flaws.

Build process

The quality of a project's build process is important for two reasons. Some organizations reuse open source code projects through binary distributions, as libraries, that they link with their other code or as components that run on their infrastructure. If your organization works like this, at some point you may need to build the binary from source code to fix a bug or add a feature required by your organization. Other organizations (mostly larger ones) have strict rules against using random binaries off the Internet and have processes for building everything internally from source (at least once).

Whatever the case, it's sensible to check how easy it is to perform a project build. Is the procedure documented? Does it work in your environment? Will you need some rarely used build tools, an unsupported integrated development environment, or a compiler for an exotic programming language? For critical dependencies, evaluate these requirements in the same way that you're evaluating the primary open source project under consideration.

PROCESS

No matter how shiny the open source project appears to your eyes, you also should invest some time to examine how it is produced and managed. This will affect your experience with it in the long term and also may uncover potential pitfalls that weren't discernible from the product's examination.

Software released under version 1.1 of the Mozilla Public License cannot be linked together with code licensed under the GNU General Public License.

Development process

Start by evaluating the quality of the project's development process. Does the project practice continuous integration? You can easily determine this by looking for corresponding configuration files (for example, `.travis.yml` or `Jenkinsfile`) in the project's root directory. Examine what the continuous integration pipeline exercises. Does it, for example, include static analysis of the code as well as unit testing? Does it build and spell-check the documentation? Does it calculate testing code coverage? Does it enforce coding standards? Does it check for up-to-date dependencies? A shortcut for answering these questions are badges appearing in the project's GitHub page, though their significance is not always a given.[2]

Code commits

Then look at code commits to the project's revision management repository. Are commits regularly made by a diverse group of committers? Unless the project is very stable and likely to remain so (consider a numerical library), a lack of fresh commits may imply that nobody will step in to address new requirements or bugs. Similarly, commits by a single author or very few signal that the project suffers from a key person risk. Also known as a *bus factor*, this identifies the danger the project faces if, for example, a lead developer is hit by a bus.[3] Also, look at the details of a few commits. Are they clearly labeled and appropriately described? Do they reference any documented issues that they have addressed using a standard convention? Is there evidence that code changes and additions have been reviewed and discussed?

Project releases

Down the road, see how these commits translate into complete project releases. Are these sufficiently recent and frequent? For cutting-edge projects (say, a deep-learning library), you want to see regular updates; for more stable ones, you're looking for evidence of maintenance releases. In some cases, frequently integrating new releases of an open source component into your code base can be disruptive, due to the risks and additional work of this process. To avoid these problems, check for a separate release channel for obtaining only security and other critical fixes. Additionally, to minimize the disturbance associated with bringing in major updates, see if there are so-called long-term support releases and determine whether their time horizon matches your project's pace.

Support channels

Source code availability is an excellent insurance policy for obtaining support because it allows you to resolve issues and fix bugs within your organization; "Use the source, Luke," to paraphrase a line from *Star Wars*. Such measures, however, are typically extreme. When using open source software, a helpful support forum is usually the most practical way to resolve such problems. Consequently, look for the project's available support channels. Is there an online forum, a mailing list, or a chat group where you can ask questions? Do useful answers arrive quickly? Are respondents supportive and friendly?

A lack of fresh commits may imply that nobody will step in to address new requirements or bugs.

In my experience, the quality of a project's technology and its support are orthogonal. Some projects with mediocre quality code offer excellent support and vice versa. For enterprise scenarios where it's not prudent to rely on volunteer help for resolving critical issues, you may also wish to examine the quality of paid support options offered through specialized companies, consultants, or products.

Handling issues

Inevitably, at some point, you're likely to encounter a bug in the open source project you're using. Therefore, it's worth examining how the project's volunteers handle issues.[4] Many open source projects offer access to their issue management platform, such as GitHub Issues, Bugzilla, or Jira, which allows you to look under the hood of issue handling. Are issues resolved quickly? How many issues have been left rotting open for ages? Does the ratio between open and closed issues appear to be under control, that is, in line with the number of project contributors?

Contributing fixes and enhancements

Another scenario down the road concerns the case where you make some changes to the project's source code, either to fix a bug or add a new feature that your organization requires. Although you can keep your changes to yourself, integrating them into the upstream project safeguards their continued availability and maintenance alongside new releases (in addition to it being the proper thing to do).

Evaluate how you'll fare in this case by examining how easy it is to contribute fixes and enhancements. Is there a contributor's guide? If you're using the project as a binary package, is it easy to build and test the project from its source code? Through what hoops do you have to jump to get your contribution accepted? Is there an efficient method by which to submit your changes, for example, through a GitHub pull request? Does the project regularly accept third-party contributions? Note that some organizations release projects with an open source code license but allow little or no code to be contributed back to their code base.

All 13 evaluation criteria I've outlined in Table 1 are important, and taking them into account can spare you unpleasant surprises and the cost of switching from one project to another. Furthermore, you can use Table 1 as guidance on how crucial some criteria are in specific contexts. Specifically, those identified in the first colored column can be deal breakers. In addition, if you identify problems with the yellow-marked criteria in the second column, this means that you'll need to build in-house capacity to support the corresponding open source project. Finally, if you decide to support the project with in-house resources, then the green-marked components in the third column become more important. Ultimately, all of these checks will help to ensure a long, happy, and prosperous relationship with the open source components you're selecting for your work. **C**

ACKNOWLEDGMENTS

I thank Zoe Kotti and Alexios Zavras, who made many helpful suggestions in an earlier version of this article.

REFERENCES

1. A. Morin, J. Urban, and P. Sliz, "A quick guide to software licensing for the scientist-programmer," *PLOS Comput. Biol.*, vol. 8, no. 7, pp. 1–7, 2012.
2. A. Trockman, S. Zhou, C. Kästner, and B. Vasilescu, "Adding sparkle to social coding: An empirical study of repository badges in the *npm* ecosystem," in *Proc. 40th Int. Conf. Software Engineering*, 2018, pp. 511–522.
3. K. Yamashita, S. McIntosh, Y. Kamei, A. E. Hassan, and N. Ubayashi, "Revisiting the applicability of the Pareto principle to core development teams in open source software projects," in *Proc. 14th Int. Workshop Principles of Software Evolution*, 2015, pp. 46–55.
4. T. F. Bissyandé, D. Lo, L. Jiang, L. Réveillère, J. Klein, and Y. L. Traon, "Got issues? Who cares about it? A large scale investigation of issue trackers from GitHub," in *Proc. IEEE 24th Int. Symp. Software Reliability Engineering (ISSRE)*, 2013, pp. 188–197.

DIOMIDIS SPINELLIS is a professor of software engineering and head of the Department of Management Science and Technology, Athens University of Economics and Business. His most recent book is *Effective Debugging: 66 Specific Ways to Debug Software and Systems*. He is a Senior Member of the IEEE and ACM. Contact him at dds@aueb.gr.