Performance of Open Source Projects[1]

Michael Weiss

Carleton University, Ottawa, Canada
weiss@sce.carleton.ca

1 Introduction

The patterns in this paper describe open source development practices from a performance perspective. In product development, performance is measured in terms of the *time* it takes to develop a software product, the resulting *quality* of the software, and the *cost* of development. These dimensions are in tension with one another. Since improving performance has side effects, we also need to include the impact on other dimensions than performance (eg risk, trust) in our discussion of the practices.

The audience for these patterns are developers and project managers, who are thinking about adopting an open source development approach. The practices documented in the patterns are derived both from the literature on open source development, and from the author's experience as contributor to several open source projects. The author also had the opportunity to observe a large open source project.

2 Patterns

An open source project must start with a *Credible Promise (3)*, otherwise it will fail to attract developers. To encourage other developers to build on it, a project needs to offer sufficient functionality or equivalent.[2] For the open source project to grow fast, it is necessary to build up momentum quickly. Rather than waiting for a polished release with full functionality, the owners of the project should aim to deliver a stream of *Frequent Releases (5)*. This allows them to learn from users how the code is used; advanced users will discover and fix problems for you, resulting in higher code quality.

In order to achieve a critical mass of functionality early in the life of a project, the project should *Build On the Shoulders of Others (7)* by integrating components developed by others. Reusing components with proven functionality also allows a project to reach a higher level of quality earlier.

To keep others – users and developers of components – engaged in your project you need to maintain an *Open Dialog (9)*. Development of the system needs to be carried out in the open. At the same time, not all developers participating in a project may pursue the same goals. Developers need to be able to explore different ways of evolving the system in parallel. Hence, *Parallel Development (11)* provides a mechanism for coordinating between stable and experimental releases of the same project.

---

1 Copyright is retained by author. Permission is granted to Hillside Europe for including in the CEUR archive of conference proceedings and for the Hillside Europe website.

2 For some projects the biggest attraction for other developers is not the functionality (all that exists may be a specification), but the reputation of the project founders (Fogel, 2006).

A map of the patterns showing their relationships is shown in Figure 1. Links between patterns X and Y should be interpreted as "after pattern X you may also use pattern Y". Patterns with a star (*) are planned patterns. Thumbnails of these patterns can be found in Appendix A.

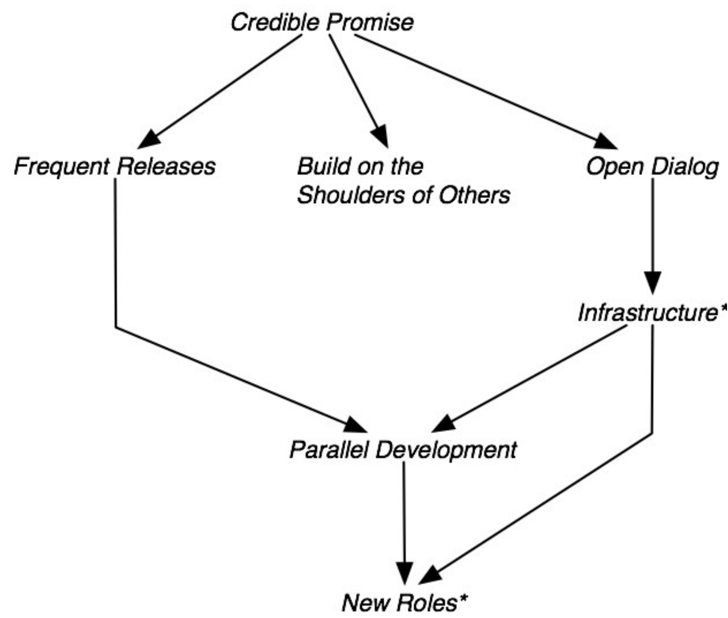

Figure 1: Patterns for the performance of open source development

2.1 Credible Promise

What both projects did have was a handful of enthusiasts and a plausible promise. The promise was partly technical (this code will be wonderful with a little effort) and sociological (if you join our gang, you'll have as much fun as we're having). So what's necessary for a bazaar to develop is that it be credible that the full-blown bazaar will exist! (Raymond, 1998)

Context You are starting an open source project. So far you have been working on the project by yourself. You want other developers to join and leverage their contributions to grow the project.

Problem **How do you mobilize developers to contribute to your project?**

In economics this problem is known as penguin problem. Hungry penguins are gathered on a floe of ice. However, none of them wants to dive first for fear of being eaten by a predator. No penguin moves until every penguin moves.

Forces If your project does not provide a core of the intended functionality, developers will not have enough incentives to join your project.

Developing an initial core of functionality takes time.

You don't want to be too far ahead in your implementation, but leave developers with unresolved challenges.

When developers join they need to make an investment in your project, something they lose if your project fails.

Sometimes you already have code from a closed source product that you want to open up. Opening up means that you need to release control. Otherwise, why would external developers work for you for free, if they didn't share ownership with you.

Therefore,

Solution **Build a critical mass of functionality early in your project that demonstrates that the project is doable and has merit.**

An exception to this rule is that projects without running code can attract developers when their creators have a high reputation.

Consequences A critical mass of functionality is valuable for other developers.

Time spent on developing the initial core to attract developers early on in the project is often offset by faster growth later. However, there is also a risk: you may not succeed in growing a critical mass of functionality, and the extra time and effort spent on architecting the project for a community of developers will be “sunk”.

When you focus on your core functionality, other developers will find it challenging to contribute additional features.

One risk is that developers who join your project may benefit more from the project than you gain from their contributions.

Opening up a closed source project can extend its life. However, other developers can now also influence its direction.

Examples

Fetchmail and Linux are the two examples Raymond refers to.

BigBlueButton is an open source web conferencing system whose functionality was sufficient for teaching our online courses, but did not yet have the bells and whistles of competing commercial products (eg recording and playback or desktop sharing).

Subversion initially had no running code. Its credibility came from its founders, the main developers of the CVS versioning system.

Related patterns

When starting an open source project you want to move fast. Rather than waiting for a polished release with full functionality aim for *Frequent Releases (5)*. This allows you to learn from users how they use your code; advanced users will also discover and even fix problems for you, resulting in higher code quality.

To simplify the bootstrap process required to produce a critical mass of functionality *Build On the Shoulders of Others (7)*. Reusing components with proven functionality also allows a project to reach a higher level of quality earlier (ie improve quality and time).

Engaging others in your project requires an *Open Dialog (9).*

Seeding (Homsky & Raveh, 2007) an online community with content ensures that people will find it worthwhile visiting.

Benefits and risks of developing a product with other companies are described in *In Bed with the Enemy* (Weiss, 2007).

Sources

Literature (Lerner & Tirole, 2002; Fogel, 2006; Haefliger et al., 2008) and the author's observation.

2.2 Frequent Releases

Frequent release cycles are both a curse and a blessing. Software developers are creating fixes and patches all the time. The downside is the developer doesn't want to do upgrades all the time. (Klawans, 2007)

Context You need to provide a working system early.

Problem **How do you move an open source project along quickly?**

Forces At the start of a project requirements are often unclear. What the user says they want may not be what they really need.

Defects can be detected early when releases are frequent. Each release provides an opportunity for feedback.

Upgrading to a new release can be disruptive to some users.

The larger a change between two versions of a project, the more challenging the new code is to integrate.

Releases may be incomplete in terms of functionality or unstable. Users have different risk comfort levels for upgrades.

When developers share their changes infrequently, they may duplicate each others' efforts by solving the same problem.

It takes time away from other tasks to keep up with changes.

Therefore,

Solution **Release code in small, quick increments.**

Don't hold off a release until the functionality is complete, but make changes to the code available as soon as the code is complete, ie when it compiles and executes. Each incremental release also brings the system closer to functional completeness.

Consequences Releasing a system in frequent increments protects you against overdesign. Each release is an opportunity to receive feedback from users that helps you shape the direction of the project.

The quality of the project increases when defects are removed early. Higher quality means that you spend less time fixing bugs.

User expectations need to be carefully managed. Releases need to be labelled as stable or development releases.

Frequent releases allow continuous integration. Each integration step is small, allowing problems to be quickly localized.

Developers can reuse intermediate releases, which can significantly shorten development time for individual features.

As changes are shared earlier, developers can lever partial solutions

by other developers, making them more efficient.

Integrating frequent releases can be difficult and time consuming. You may want to skip some incremental changes.

Examples

Linux, Apache, and Mozilla all have frequent releases.

While BigBlueButton initially did not create releases frequently enough so as to maximize development velocity, automated builds are now created after each successful commit (that passes tests).

Related patterns

Working with external developers, who may pursue different goals while they share an interest in your project, requires a mechanism for coordinating between stable and the experimental releases made by different developers, as described in *Parallel Development (11).*

Incremental Integration (Harrrison & Coplien, 2006) ensures that subsystems work well together. It encourages developers to check regularly for incompatibilities with other subsystems.

The technical implications of frequent releases (out of scope for us) are described in *Continuous Integration* (Elssamadisy, 2007).

Sources

Literature (Lui & Chan, 2008; Fogel, 2006; Harrison & Coplien, 2006; Goldman & Gabriel, 2005), and the author's observation.

2.3 Build On the Shoulders of Others

We used open source extensively in the creation of the Nokia 770. We favored components that were developed by active communities and already used by many. [...] We created the product in shorter time and with fewer resources, compared to other products utilizing proprietary software. In essence, open source offers time and cost savings in a form of readily available components and subsystems, available developers, and effective development model. (Jaaski, 2006)

Context You need to build a critical mass of functionality.

You need to provide a working system early.

Problem **How do you grow a critical mass of functionality quickly?**

Forces The faster you can deliver the core functionality, the more and the sooner you will be able to attract other developers.

Reuse allows to leverage the code and experience of others, but it takes time to select and to understand external code.

Not invented here (NIH) prevents us from looking outside. Personal pride can be in the way of reuse.

Code that you reuse is not streamlined to your project goals.

You only own the intellectual property (IP) on code that you write yourself, or that you paid somebody external to develop.

Maintaining your own code is hard enough. Maintaining external code of potentially variable quality can be much harder.

Not all pieces of your system are equally valuable. Building all pieces yourself means that you will need to spend time on building functionality that you would rather spend elsewhere.

Therefore,

Solution **Integrate assets from other open source projects.**

Your development time is reduced by leveraging the functionality provided by existing code and building on existing designs. Your main task becomes one of writing “glue” code that links these code assets. Much of your leverage will come from building on code

developed by others in the form of libraries, components, or tools. However, there are other opportunities for reuse: APIs (a new implementation of an existing API), exchange formats (writing to and reading from existing formats), services (invoking code that is hosted elsewhere), requirements (cloning another design), and test suites (compliance with an existing specification).

Consequences

You shorten the time to deliver a critical mass of functionality, which provides an incentive for others to join your project.

By reusing proven code, you can also produce code of higher quality, sooner. However, this introduces a new problem: your project now becomes dependent on other projects you don't control. Whenever one of the projects you depend on changes, you need to incorporate them back into your project, otherwise you will not be able to benefit from bug fixes made to those projects.

Your ability to reuse depends on how well you have adopted a culture of reuse. Reusing existing open source components is not necessarily "second nature" to your developers. It may help to make reuse part of refactoring by putting a process in place that ensures that quick initial solutions are replaced by existing components for longer-term stability of the project.

You inherit "baggage" from existing assets that you don't need, and which may lead to duplication and to maintenance problems.

Also, as you reuse code created by others, you need to ensure that you comply with the licenses associated with this code.

You need to maintain a stack of software artefacts developed by others. But this also creates opportunities for third parties who take on the task of managing the stack (companies like RedHat).

Reuse allows you to focus on the parts you like to work on, or which add most value by incorporating existing components for the parts that are necessary, but less interesting or valuable to you.

Examples

The initial version of BigBlueButton was built in a few months by combining many existing pieces of functionality, including Red5, an open source version of the Flash Media Server; Asterisk, an open source PBX; OpenOffice to convert PowerPoint slides to PDF; and Apache ActiveMQ, an open source message broker. The actual code of the initial version was only around 10K lines of code.

Many projects (both small and large) build on the Eclipse platform by developing their application as an Eclipse plugin. There are more than a 1000 plugins listed on the Eclipse Plugin Central site. The code common to the plugins is provided by Eclipse.

OpenOffice is both an example of requirements cloning and reuse of exchange formats. OpenOffice implements most of the feature set of Microsoft Office, and can read and write Word documents.

Related patterns

Leveraging existing code to speed up development is suggested by *Prototype a First-Pass Design* (Foote & Opdyke, 1994).

There are different ways of achieving license compliance. Patterns have been documented for licensing, but not for compliance.

Sources

Literature on reuse in open source (Haefliger et al., 2008), software cloning (Lui & Chan, 2008), and the author's observation.

2.4 Open Dialog

The open-source approach is new to Autodesk, especially publicly discussing new features. It has taken considerable effort to get used to this, but we have taken it to heart, and all new features for MapGuide Open Source are debated in the public mailing list before we start development." (B. Dechant, MapGuide technical architect), quoted in Birch (2007)

Context You want users and other developers to contribute to your project.

Problem **How do you engage others in your project?**

Forces You need to listen to your users; the dialog must be two-way.

You don't want to appear weak by sharing your ongoing decision process, including the wrong turns and mistakes.

It is difficult to guess what users really want. If you ask them, they are often unable to articulate their needs.

Users and external developers need to feel valued.

You can't build everything yourself.

Other users and developers must benefit from participating.

Therefore,

Solution **Conduct the project in the open, maintaining a two-way dialog with project participants (users and external developers).**

Give users and external developers access to your source code. Artefacts to share include the code, installation instructions and binaries (as applicable), and test plans. Allow outsiders to participate in your project decisions by discussing project plans on a wiki or a project mailing list. Create opportunities for others to participate by allowing them to fix bugs and add features.

Consequences Users will provide you with valuable feedback. This allows you to learn from your users and improve the "fit" with their needs.[3]

Making your decision process transparent builds trust between you and other project participants.

When you share your decisions with your users, they can tell you that what you propose to do is not what they want.

Openness encourages reciprocal behavior. Sharing with others will

3 Quality depends as much on building the right system (meeting explicitly stated requirements) and building the system right (meeting the unstated requirements as well, ie fit).

cause them to contribute back to your project.

You will be able to achieve more by enlisting outside contributions.

The result of open participation is greater value than what any individual contributor could have achieved on their own.

Examples

The contributors to the BigBlueButton project are developers at the company co-founded by the original developer, students, faculty of the university (who are also lead users of the system), and members of other businesses who are developing complementary products. Code, project plans, as well as bugs and feature requests are shared by hosting the project on Google Code. Currently, any registered contributor can contribute equally to the project.

MapGuide open source project by Autodesk quoted above.

Hosting sites such as SourceForge or Google Code provide a set of tools for publicly sharing the output of an open source project.

Related patterns

Maintaining an open dialog requires a supporting *Infrastructure (13)* of tools such as wikis, mailing lists, bug trackers, or repositories.

As a project grows, tasks such as moderating a mailing list require dedicated resources. The answer is to create *New Roles (13)*.

Canned hosting sites like Google Code provide you with a technical *Infrastructure* to maintain an open dialog. These sites typically include code repositories, wikis, mailing lists, and bug tracking tools. They also maintain developer profiles and project statistics.

Engage Customer (Harrison & Coplien, 2006) ensures that there is a continual exchanges between developers and customers.

Gatekeeper and *Firewall* (Harrison & Coplien, 2006) deal with the issues of translating and filtering external interactions.

Sources

Literature on open participation in open source projects (Goldman & Gabriel, 2005) and the author's observation.

2.4 Parallel Development

The current production versions are Python 2.6.2 and Python 3.0.1. You should start here if you want to learn Python or if you want the most stable versions. Note that both Python 2.6 and 3.0 are considered stable production releases, but if you don't know which version to use, start with Python 2.6 since more existing third party software is compatible with Python 2 than Python 3 right now. (Python Software Foundation, 2009)

Context You need to manage the expectations of your users.

Problem **How do you balance the need of users for stability with the need to explore new directions for your project?**

Forces While developing the current release of your system, you also need a way of working on new features for future releases.

External developers may pursue different goals from you when they participate. You need to give them a mechanism for pursuing their interests, while benefiting from their contributions.

Therefore,

Solution **Maintain separate release streams, those with the official stable releases, and others for experimental development.**

There can be multiple levels of stability (such as nightly builds, weekly releases, and scheduled milestones).

Consequences This solution addresses the needs of your internal development (feature roadmap), as well as those of external developers. The stable versions of your project can be used for ship products, and experimental versions allow you to explore future products. As you overlap maintenance and new feature development, you also shorten the time to introducing those new features.

However, developers need to allocate time to coordinate between the different versions, which they cannot use to write new code. For example, when a bug is fixed in the stable version, it needs to be applied to all the experimental versions as well.

Examples The BigBlueButton project has separate streams for the production version, which is used to teach online classes and therefore requires the behavior of the system to be stable, and streams for new feature development that will eventually be rolled into the stable stream.

The Eclipse project has a yearly release train. Projects to be contained in the release train need to meet a set of well-defined

requirements (such as signed-off milestones).

The Python project currently has two stable releases (see quote).

Related patterns

In a large project managing the different releases requires dedicated release managers, and example of *New Roles*.

Named Stable Releases (Harrison & Coplien, 2006) provide a handle for communicating changes to developers.

There are many patterns such as Berczuk (2003) about the technical aspects of configuration management (out of scope for us).

Sources

Literature on parallel development in open source (Fogel, 2006; Muffatto, 2006; Davies, 2009), and the author's observation.

3 Conclusion

In this paper we presented a first set of patterns on open source development that will form the core of a larger pattern language. In this set we focused on open source practices that improve the performance of a project, that is, how adopting these practices helps reduce the time, improve the quality, and reduce the cost of an open source project.

There are several parallels between open source and agile development practices as noted by Lui & Chan (2008) and Goldman & Gabriel (2005). This paper does not claim that practices like *Frequent Releases (5)* and *Parallel Development (11)* are unique to open source development, but it emphasizes their key position in the open source paradigm. Other practices related to making development transparent and creating a credible promise are more germane to open source development. However, every practice described has aspects unique to the context of open source development.

Future papers will document patterns for other perspectives on open source development. Overall, the author envisions the collection of patterns to contain patterns on the strategic use of open source (why adopt an open source approach), open source product development (of which these patterns are a part), technical architecture (how are open source systems structured to encourage contributions), licensing aspects (impact of license choices), and governance of open source projects (organization).

Acknowledgements

I thank Cecilia Haskins for shepherding this paper. I especially thank her for her deep insights into patterns and her patience with a slow author.

In formatting these patterns I owe a tremendous amount to the format Allan Kelly has used in his papers, which I tried to emulate.

Appendix A – Planned patterns

Here are short forms of the patterns not described in this paper.

Infrastructure	How do you share project decisions effectively? Lever coordination tools that can be accessed by all project participants. These include mailing lists, messaging, code repositories, bug tracking tools, and wikis.
New Roles	When a project gets too large, how do you coordinate contributions? Define formal roles for contributors (such as issue manager) and the interaction between them.

Appendix B – Contributions of the patterns

This table summarizes how the patterns described in this paper affect the different dimensions of performance (time, cost, and quality). These relationships could provide the basis for testable hypotheses for a future study, which may empirically establish the impact of the patterns.

	Minimize Time	Minimize Cost	Maximize Quality
Credible Promise	+		
Frequent Releases	–	(–)	+
Build on the Shoulders of Others	–	(+)	+
Open Dialog		(–)	+
Parallel Development	–	(–)	+

References

I tried to limit the number of references, but the ones below are needed to give proper attribution. Key references are highlighted with a (*).

Baldwin, C., and Clark, K. (2006), Architecture of participation: does code architecture mitigate free riding in the open source development model?, Management Science, 52(7), 1116-1127.

Berczuk, S. (2003), Software Configuration Management Patterns: Effective Teamwork, Practical Integration, Addison Wesley.

BigBlueButton (2009), http://code.google.com/p/bigbluebutton.

Birch, J. (2007), MapGuide Open Source: Project Insights and Practical Applications, August, Geoplace.com

Davies, T. (2007), On branching and frequent releases, http://twmdesign.co.uk/theblog/?p=37

Elssamadisy, A. (2007), Patterns of Agile Practice Adoption, InfoQ.

* Fogel, K. (2006), Producing Open Source Software: How to Run a Successful Free Software Project, O'Reilly.

Foote, B., & Opdyke, W. (1994), Lifecycle and refactoring patterns that support evolution and reuse, PloP, and in Coplien, J., & Schmidt, D. (1995), Pattern Languages of Program Design 1, Addison Wesley, http://www.laputan.org/lifecycle/lifecycle.html.

* Goldman, R., & Gabriel, R. (2005), Innovation Happens Elsewhere: Open Source as Business Strategy, Morgan Kaufmann.

* Haefliger, S., von Krogh, G., & Spaeth, S. (2008), Code reuse in open source software, Management Science, 54(1): 180-193.

* Harrison, N., & Coplien, J. (2006), Organizational Patterns of Agile Software Development, Addison Wesley.

Jaaski, A. (2006), Building consumer products with open source, Linux Devices, Dec, http://www.linuxdevices.com/articles/AT7621761066.html.

Johnson, R. (2007), Is it a tomcat, or the elephant in the room, Spring Source blog, posted on Dec 24, 2007, http://blog.springsource.com/2007/12/24/is-it-a-tomcat-or-the-elephant-in-the-room.

Lerner, J., & Tirole, J. (2002), Some simple economics of open source, Journal of Industrial Economics, 50(2): 197–234.

* Lui, K.M., & Chan, K., Software Development Rhythms: Harnessing Agile Practices for Synergy, Wiley.

Milinkovich, M. (2008), A practitioner's guide to ecosystem development, Open Source Business Review, October, www.osbr.ca.

* Muffatto, M. (2006), Open Source: A Multidisciplinary Approach, Imperial College Press.

Raymond, E. (1998), interviewed by F. Cavalier from Mib Software.

Homsky, O., & Raveh, A. (2007), Pattern language for online communities, EuroPLoP.

Weiss, M. (2007), In bed with the enemy, EuroPLoP.

Photo credits

Penguins on an ice flow, by T. Ellis and shared under a CC-BY-NC license, http://www.flickr.com/photos/tim_ellis/26360944

Human pyramid, by somerandomsequence, shared under a CC-BY-SA license, http://www.flickr.com/photos/somerandomsequence/3898247588

Synchronize, by sammiji, need to obtain rights or replace, http://www.flickr.com/photos/sammiji/3417689614

Open door, by D. Seagers and shared under a CC-BY-NC-SA license, http://www.flickr.com/photos/seagers/1805045379

II, by Lori B. and shared under a CC-BY license, http://www.flickr.com/photos/13025462@N08/2905698081