Improving Open Source Software Development with an Object Oriented Design Model

Joshua Kugler
Software Engineering/SWE 471W Fall 2003
University of Alaska Fairbanks
PO Box 750436
Fairbanks, AK 99775

Abstract. Open Source Software (OSS) is a phenomenon that has been gaining popularity in the past few years. In it, the source code to a software project is made publicly available for all to examine, tinker with, and change. While this has the advantage of making it possible for contributions to come from far and wide, OSS has not reached its full potential. This has been mainly due to small groups, unclear design, and poor documentation, which results in a project that is difficult to understand. Because of this, recruiting developers is challenging. A better system of development is needed, and we will propose that the better system is that of an Object Oriented Design model. We will explain what OOD is, show why it is better, what must be in place to take advantage of it, and show the improvements that can result.

I. What is Open Source Software?

According to the Open Source Definition (Perens 1997), “Open Source” denotes a software license which allows for the unlimited modification and distribution of, as well as derivation of other works from, the part of the software to which the license applies. There are currently over 45 licenses certified by the Open Source Initiative as following the Open Source Definition.

Freely available software has been developed for a few decades, going back to the early days of Unix, when programmers shared code and ideas. But only recently has the term “Open Source” come in to use to describe software that is freely available and freely distributable, along with its source code. The Open Source movement came into its own with codification of what people knew, and had been doing, for many years.

In the “The Cathedral and the Bazaar” (Raymond 1997), Eric Raymond substantiated the benefits of the Open Source process by relating the history of, and his experiences with the mail retrieval client “Fetchmail.” This paper was, and probably still is, the most lucid explanation of why the Open Source process works so well. About the time this paper came out, there was a large meeting of the free software community. At this meeting, it was decided that “free software” needed a new image. The term “free software” had acquired somewhat negative connotations, so a new name was needed to describe the true idea behind the philosophy. This new name was “Open Source” (DiBona 1999).

II. The Most Common OSS Model

As documented in (Krishnamurthy 2002), even mature OSS projects do not have the team model we'd expect, nor the model that is touted as the “open source ideal.” Instead of being teams of “hundreds or even thousands, (Mockus 2000)”—a number that is commonly pulled out of the air when describing the joys and merits of OSS development—they are often small teams. In fact, the average number of developers in 100 mature OSS project was not even 7, with the most common being only one. The average numbers found for project administrators were similarly low: just over two, and again, the most common was just one (Krishnamurthy 2002).

There are, however, exceptions to this general pattern, two of which are the Linux kernel and the Apache web server. The simplest reason for their status of “exception” is that they do not fit the above descriptions. A brief discussion will be given here.

The Apache web server project, as of 2000, had 400 contributors, with 15 of those people being among the top developers who contributed nearly 90% of the code (Mockus 2000). This does not include individuals who contributed bug reports or other quality-related assistance. The Apache project is run on a committee rule basis. This committee is a meritocracy, and any changes that are not unanimously agreed upon require a vote by a quorum of the committee (Mockus 2000).

The Linux kernel could be said to have hundreds of developers as well. As of the latest kernel at the time of this writing (2.6.0-test9), there are 447 people acknowledged in the CREDITS file. There are two primary reasons for this. First, it is a large open source project, so is going to attract many developers. These developers will have a desire to improve the project, (for both pragmatic and personal reasons). Second, many people only work on small areas, usually device drivers and such. But unlike the Apache project, Linux has a hierarchical leadership structure, with Linus Torvalds at the top as the “‘benevolent dictator.’ This leads to a situation where there are parts of Linux with their own ‘benevolent dictators,’ although their ‘power’ is limited by the fact that [Linus] will always have the last word. (Robles 2003)” The modular architecture of the kernel has allowed a very modular model for the leadership structure.

III. Problems With the Prevalent Model

One of the major downfalls of these small “single-man” (or few-man) projects is that there is not much documentation. When a project only has one programmer, that individual will spend most, if not all, of his or her available time working on the code of the product, and not on other peripheral issues; in this case, documentation. Thus, documentation can often be sorely out of date, or nonexistent. The lack of documentation in many open source projects is acknowledged (Ferraz 2003) . Based on experience (Kugler 2002, and others projects), the author can speak to the tedium and bore of writing out what the program does and how it does it. It is much more enjoyable, and ostensibly more productive, to simply write code and get the problem solved and implemented. Even more onerous than documenting the program is keeping those documents in sync with the current program. Especially when a program is in rapid development, it can be a very difficult, and at times discouraging, game of catch-up to make sure the documentation accurately reflects the current state of the code.

A second area of concern commonly encountered is that these small project are not organized very well when it comes to the layout and design of the overall software product (Foote 1997). Most projects are started by programmers who know how to write good code, but who do not have a strong grasp on software engineering concepts. They will not be familiar with common, and necessary, software process models, so the organization of both the development, and the project itself, will be very unstructured. In addition, there may not be a strong emphasis of functionality or interface/implementation/user interface separation. Thus, the entire project is one large tangled series of code with many issues which complicate modularity, such as user interface manipulation outside in the core of the implementation, instead of true separation.

This, first of all, makes the code very hard to document, as the documenter has to reference code in another other file (or worse yet, in other places in the same file), instead of simply explaining the functionality that is called in another module. In addition, it will be most likely impossible to delegate sections of the code to others, as there is no clear-cut way to delineate the sections in which the subordinate should work. Most of all, though, it is non-trivial for a newly integrated developers to understand the code, since there is no logical order or layout to the project (Foote 1997). This, in turn, makes it very difficult to recruit new developers, as they will discourage quickly when they cannot easily understand the code and the overall project.

IV. An Improved Process is Needed

An improved approach is obviously needed. This improved process is Object Oriented Software Design, or stated more unambiguously, Object Oriented Design applied to Open Source Software. OOD works by breaking projects into objects, "refining candidate objects into classes, defining message protocols for all objects, defining data structures and procedures, and mapping these into an object-oriented programming language (Bray 1997)." OOD is a design model that requires explicit conceptualization, design, and abstraction of the goals (the end product desired); architecture; modules; intra-application communication protocols (e.g. how to communicate with other parts of the program); and inter-application communication protocols (e.g. how an external program can query/user functionality from the program).

V. Why OOD is Better

The OOD approach has two advantages. First, it is straight forward to gain understanding of the project in discrete units: for example, component by component or module by module. Second, delegating responsibility for various components now logically follows from structure because of the separation inherent in the system.

The author can speak to the accuracy of point one above. Over this past summer, he was involved in the reimplementation of a web site using a content management system (CMS). In his investigations, he came across a system called Metadot (See *Sites*, below). Metadot is an open source CMS produced by the Metadot Corporation. This system had all the features needed, save a couple. It was modular, could easily assign permissions to areas of the web site, and was easy-to-use for a user without much technical expertise. What it did not have was a robust calendar module and a photo gallery.

Extending the system was very logical, and after less than a week of reading documentation and playing around with the system, he was able to add the needed functionality (See DoHGallery, *Sites*, below). This was because the Metadot CMS has a very clearly defined and documented plug-in/modular architecture that can be used to quickly add features, or otherwise modify default behavior, without having to touch or dig through unrelated code.

As to point two, when parts of the project grow large enough to require their own oversight and coding/maintenance team, it will be very straight forward to give write permissions to small, discrete areas of code. Trying to coordinate the needed changes in a large monolithic system will be difficult once the size of the code grows beyond a couple thousand lines. Similarly, when a new developer expresses an interest in a modifying one of the components, he or she can be given free rein in that module, and will not have to interfere in the development of, or even understand, the other parts of the program, as long as he or she understands the given module and its documented interface.

VI. Requirements to Apply OOD to OSS

The first requirement that must be in place for OOD is a precise road map of where the project is going: goals, intentions, and desired product. It must be documented from the outset, and known by all, what is the intended outcome. A project which bypasses the design step and proceeds straight to implementation will not be well structured, stable, or robust. And most of all, it will be very difficult to track down bugs and understand the overall system. For small projects, this road map can be as simple as a list of desired features, but ideally, it should be a list of explicit, unambiguous goals and requirements which will facilitate the analysis and design of the software system in a modular, object-oriented way.

Second, there must be a well documented, systematically derived architecture for the product, and a systematic structure definition for the entire project. This requires explicit design (planning in advance) instead of implicit design (planning as the code is typed). For example: the planned feature set of the product, as well as how it will be broken down into its requisite components, needs to be fully defined before any code is written.

Because of changing user requirements and the diversity of user requirements—especially in an Open Source product—even with a well defined structure definition, flexibility in the feature set of the product is still desired, even needed. Unanticipated needs and features will arise, and there must be a way to integrate these *ex post facto* design considerations into the overall software architecture. Flexibility can still be found, even in a rigid design structure, and can be found (at least) two ways. The first way to keep flexibility is to add new features within the current architecture and design guidelines. For example, if one wants to add another dialog box, then it will be integrated into a logical place (logical being defined by the overall design) on the menu. The software architecture will also define where in the code tree the new code (both interface and implementation) will reside. It also needs to communicate with other areas of the program via defined calls to classes that handle the duties of those other areas. The second way to retain flexibility is to design the software with a "plug-in" architecture. This will allow new modules and functionality to be added to the system, possibly even at run-time, without interfering with existing code.

Third, there must be precisely defined connectors for communication within the program and between the program and other systems. No part of the program can directly manipulate or read the data structure of another component. A class of that component will be instantiated, and then the desired data read or written. For example, no part of the program will directly read (or write) the preferences database. Instead, it will instantiate a preferences class, and then request the parameters it requires (and write the parameters it needs to save). Similarly, no part of the program will write directly to records on disk; using instead disk I/O class(es) for this purpose. This also applies to programs which use a database back-end to store their data. In this case, all communication will go through a custom database I/O class. This will allow changing the back-end database by only modifying code in one class in the project.

Precisely defined connectors and protocols are especially crucial in Open Source projects. Most communication in OSS projects is not face-to-face. E-mail lists and online forums are the norm. Any ambiguity in the definition of connectors or instability in their operation will have a drastic effect on the efficiency of the developers, as time will be spent trying to clarify definitions and protocols instead of implementing the dictated design.

Fourth, there must be leadership. Whether this is the (possibly multilevel) "benevolent dictator" model of Linux (Robles 2003) or the committee model of the Apache project (Mockus 2000), there must be a clearly defined hierarchy of command. Patches to the project, and changes to the existing architecture, must go through approval and review. But more importantly than this is the fact that someone must decide to whom a part of the project will be delegated. This will be very difficult if the project is monolithic, and the changes must be made in several places. If this is the case, then the trust afforded the developer must be much higher than if he or she was simply given the ability to change a single, discrete component. When the system is modular, the changes will be confined to certain modules, and any detrimental changes can be more easily rolled back.

VII. Benefits of OOD applied to OSS

Object oriented design requires a new paradigm, and most likely a relearning of procedures used to properly design and implement products. But, as we will see, learning OOD, and enforcing its structure will allow groups to reap several benefits.

As we saw above, non-object oriented projects can be difficult to document. As a first benefit of OOD, we will have a program that is more straight forward to document. When components of the program are instantiated, a simple comment can explain why they are being instantiated. When functions from those classes are called, another simple comment will explain the reason there too. In a more non-modular project, there will often be code that is very redundant (repeated in several locations). This redundant code makes the flow of the program harder to understand, and difficult to document.

Another benefit will be that each module can be documented separately. The modules, and their interdependencies can be documented. Then, when a modules calls another module, a line in the documentation can simply explain so, probably with a link to that section in that module's document. It does not have to explain in detail, for example, how it opens the preferences database, and finds the data it wants. This will improve the efficiency of the developers as well, since he or she will only need to understand his or her own module. This will save them time, he or she will not have to understand the entire project to improve the functionality of a single module in the project.

Non-object oriented projects can be hard to understand. A second benefit of OOD will be the greater ability to quickly understand the project component by component, and as a whole, versus a non-modular project. As one walks through code in an attempt to understand it, one line comments on calls to other modules can provide enough detail to understand what is going on locally, but will obviate the need to explore the called component right then and there. That can be saved for later when more detail is desired, and/or the first module is thoroughly understood. Again, this will save time, and improve the efficiency of the developers.

The third benefit will be less difficulty in recruiting developers. The author can speak to the discouragement of not understanding how a system works, even after much reading and code exploration. When the project is clearly modular, the advantage described above will hold, and a new developer coming into the project will easily be able to understand it as a whole, or maybe only learn the part he or she has an interest in modifying, and will be much more inclined to join the project and contribute. Recruiting and retaining developers is crucial in an OSS project, and OOD will assist greatly in this area.

A fourth benefit will be simplified bug tracking and fixing. Finding and fixing bugs in a minimum amount of time and effort is important in any kind of software project, but even more so in an OSS project. One wants to retain users, so glitches must be fixed quickly. When given functionality is confined to a single module of a program, it makes it much easier to track down a bug because it can *only* be in that module. If an error occurs in code that has been redundantly replicated throughout a non-modular project, then all those places must be examined and fixed, or else the bug may manifest at a later time. When a given procedure is only found in one module, then only that module must be fixed, and it will be fixed program wide. Also, it makes is possible to trace erratic behavior to incorrect "connector usage:" the called module may not have any bugs, but the calling module may not be passing the correct (or legal) parameters. When there is a clearly defined and documented architecture, finding these kinds of bugs becomes much easier.

The fifth benefit is the relative simplicity of extending the structure or feature set of a project at a later date, versus a large, non-modular project. As was discussed above, flexibility will be inherent in a properly designed modular architecture project. Thus, when it is time to extend a product, it will be well defined as to how the new component will be added to the system. This is especially useful in OSS projects, due to the previously mentioned diversity in one's user base. For a construction example: it is much easier to add on to a house that was built correctly in the first place. We know we have a strong foundation, and we know that any walls to which we connect will bear the loads they need to bear. Similarly, if a project is clearly broken down into its requisite sub-parts, it will be much easier to figure out where the addition will go, and the added component can use functionality already present in the program. The Apache web server project is a good example of this. If all the modules available as add-ons were integrated into the main code-base, it would be an unmanageable and incomprehensible pile of code. As it stands however, it is very easy to add functionality with a module, and functionality in other modules is available to the new modules written for the web server. In fact, Apache goes to the extreme: even the core of the web server (in a file called core.c) is a module itself: it is simply a module that is always included by default in the compile.

VIII. Conclusion

As we have seen, there are many advantages to breaking up a project into logical, smaller pieces. These enable easier documentation, easier comprehension and easier delegation. When these principles are applied to an open source project, we have the added advantage of easer developer recruitment. Also, since OSS projects are so often in a state of flux, the relative ease with which we will be able to add and reuse functionality will prove invaluable to a project.

These principles, applied to an OSS project, will result in a community of developers that is more informed, more cohesive, and more productive than any other project is that simply in the implementation stage from the start. Design takes time and thought, but the results and benefits are more than worth the extra work.

Credits and Thanks

Thanks to my CS471 (Introduction to Software Engineering) professor Mario Kupries. The topic of OOD applied to OSS was his suggestion. It was a welcome idea after I realized there was no way I would have time to research and report on the design model of the Linux kernel, and how it has changed since Linux's inception. Reading and synthesizing ten years of LKML (or even Linux Kernel Traffic) was not going to fit into my schedule.

Special thanks to everyone listed in the References section. Your excellent papers on OSS, in general, and certain projects, specifically, proved to be an invaluable resource

About the Author

Joshua Kugler is a student at the University of Alaska Fairbanks (UAF), and is currently pursuing his Master of Sciences in Computer Science. His computer related interests include databases, Perl, system administration, and online/Internet/electronic voting. He has written an Open Source online voting system for the UAF student government, which is currently in use (see the "Sites" section below for a paper on this system). He is not an Open Source zealot, but will quickly admit to being an Open Source proponent, and will push for the use of open source technologies wherever and whenever feasible. He will also quickly relate to you the joys of living in a "Microsoft-free microcosm:" his work and home computers all run Linux. He has contributed to the Metadot Open Source Content Management System, and is currently trying to find time to contribute more, and to other projects.

References

Bray, Mike, *Object-Oriented Design*, Carnegie Mellon/Lockheed Martin, October 1997 http://www.sei.cmu.edu/str/descriptions/oodesign.html

DiBona, Chris; Ockman, Sam; Stone, Mark, *Open Sources: Voices from the Open Source Revolution, Introduction.* O'Reilly & Associates, Inc., January, 1999, http://www.oreilly.com/catalog/opensources/book/intro.html

Ferraz, Ronaldo M., "Open Source and Documentation." (And attached comments) http://www.reflectivesurface.com/weblog/archives/2003/02/10/open_source_and_documentation

Foote, Brian, "The Big Ball of Mud Model." January, 1997, Department of Computer Science, Washington University. Fourth Conference on Patterns Languages of Programs http://www.laputan.org/mud/mud.html

Krishnamurthy, Sandeep, "Cave or Community? An Empirical Examination of 100 Mature Open Source Projects." Business Administration Program, University of Washington, May, 2002, http://opensource.mit.edu/online_papers.php

Kugler, Joshua, "10,001 (Or More) Polling Stations: Voting Virtually @ UAF" Associated Students of the University of Alaska Fairbanks, December, 2002 http://asuaf.org/~joshua/papers/voting_virtually.pdf

Mockus, Audris, et al, "A Case Study of Open Source Software Development: The Apache Server." Proceedings of the ICSE'2000, 2000, http://opensource.mit.edu/online_papers.php

Perens, Bruce, et al, "The Open Source Definition." The Open Source Initiative, June, 1997, http://www.opensource.org/docs/definition.php

Raymond, Eric, "The Cathedral and the Bazaar." May, 1997 – August 2000 http://www.catb.org/~esr/writings/cathedral-bazaar/

Robles, Gregorio and González-Barahona, Jesús M., "Free Software Engineering: A Field to Explore." Universidad Rey Juan Carlos, Upgrade, August, 2003, http://opensource.mit.edu/online_papers.php

Torvalds, Linus, Linux Kernel Credits File. Kernel 2.6.0-test9, 2003 http://www.kernel.org/

Sites Referenced

Metadot Corporation – http://www.metadot.com

DoHGallery – http://www.metadot.com/metadot/index.pl?iid=3197
The code for the calendar display module referenced above is not public due to use of code from the proprietary calendar system to which it interfaces.

Abstract. Open Source Software (OSS) is a phenomenon that has been gaining popularity in the past few years. In it, the source code to a software project is made publicly available for all to examine, tinker with, and change. While this has the advantage of making it possible for contributions to come from far and wide, OSS has not reached its full potential. This has been mainly due to small groups, unclear design, and poor documentation, which results in a project that is difficult to understand. Because of this, recruiting developers is challenging. A better system of development is needed, and we will propose that the better system is that of an Object Oriented Design model. We will explain what OOD is, show why it is better, what must be in place to take advantage of it, and show the improvements that can result.