Why Not Improve Coordination in Distributed Software Development by Stealing Good Ideas from Open Source?

Audris Mockus
Avaya Labs Research
233 Mt Airy Rd
Basking Ridge, NJ, USA 07920
audris@avaya.com
+1 908 696-5608

James D. Herbsleb
Bell Laboratories
2701 Lucent Lane
Lisle, IL, USA 60532
jherbsleb@lucent.com
+1 630 713-1869

Software projects are increasingly distributed among many sites, often located at great distance, both geographic and cultural, from one another. This creates the potential for enormous problems, whose effects run the gamut from enormous cumulative delay through complete breakdown and failure [1].

Open source projects are remarkable in that they usually must solve all the problems of distributed development, using only very simple communication tools such as e-mail, listservs, newsroups, and change management systems such as CVS or Bugzilla.

In a case study of the Apache server [3], the authors identified several techniques used to coordinate the work. These techniques can be summarized as follows:

- A small, elite team of capable developers, each with distinctive expertise, all of whom have commit privileges. Each is trusted by the others to make changes to the server code.
- The core team coordinates their work informally, by informing others about what they are doing, asking recognized experts in the group when in doubt, and by reviewing all changes to the code.
- In order to join the core group, candidates must clearly demonstrate competence, commitment to the work, and nearly always develop a needed specialty.
- The core group creates the vast majority of new functionality.
- A much larger group submits bug fixes. Proposed fixes are reviewed and acted upon by the core group. Bug fixes generally have fewer interdependencies than new functionality, since most of the work is usually finding the problem.
- A much larger group yet tests the code, through actual use, and submits problem reports.
- There is no formal requirements process – the requirements are determined implicitly, as whatever the developers actually build. Presumably, features are selected based on what individual developers themselves need in the product.
- Work is not assigned; individuals choose what work they will do. The choices are constrained, however, by various motivations that are not fully understood. For example, it can be assumed that developers try to maximize the chance that their code will be included in a release, and will enhance their reputation.

These techniques, as effective as they indisputably are for open source development, have certain limitations. Among them are the following:

- The core group cannot exceed some maximum size, say 15, or the overhead for the informal style of coordination will become overwhelming.
- The largest system that can be built is constrained by the size of the core team. If the size-limited core team cannot develop sufficient new functionality, additional means of coordination (e.g., formal inspections, code ownership, process, projects broken into smaller subprojects) will become necessary.
- The developers must be users, since there is generally no requirements gathering, and the developers are assumed to be domain experts. In general, only what this group of user-developers wants will actually get built.

Commercial development, on the other hand, typically uses a number of coordination mechanisms to fit the work of each individual into the project as a whole (see, e.g., [1, 2]). Explicit mechanisms include such things as interface specifications, processes, plans, staffing profiles, and reviews. Implicit mechanisms include knowledge of who has expertise in what area, as well as customs and habits about how things are done. In addition, of course, it is possible to substitute communication for these mechanisms. So, for example, two people could develop interacting modules with no interface specification, merely by staying in constant communication with each other. The "communication-only" approach does not scale, of course, as size and complexity quickly overwhelm communication channels. Ad hoc communication is always necessary, however, as the default means of overcoming coordination problems, as a way to recover if unexpected events break down the existing coordination mechanisms, and to handle details that need to be worked out in real time.

Apache adopts an approach to coordination that seems to work extremely well for a small project. The server itself is kept small. Any functionality beyond the basic server is added by means of various ancillary projects that interact with Apache only through Apache's well-defined interface. That interface serves to coordinate the efforts of the Apache developers with anyone building external functionality, and does so with minimal ongoing effort by the Apache core group. In fact, control over the interface is asymmetric, in that the external projects must generally be designed to what Apache provides. The coordination concerns of Apache are thus sharply limited by the stable, asymmetrically-controlled interface.

The coordination necessary *within* this sphere is such that it can be successfully handled by a small core team using primarily implicit mechanisms, e.g., a knowledge of who has expertise in what area, and general communication about what is going on, who is doing what, when. When such mechanisms are sufficient to prevent coordination breakdowns, they are extremely efficient. Many people can contribute code simultaneously, and there is no waiting for approvals, permission, and so forth, from a single individual. The core people just do what needs to be done. The Apache results show the benefits in speed, productivity, and quality.

The benefit of the larger open source community for Apache is primarily in those areas where coordination is much less of an issue. While bug fixes occasionally become entangled in interdependencies, most of the effort in bug fixing is generally in tracking down the source of the problem. Investigation, of course, cannot cause coordination problems. The tasks of finding and reporting bugs are completely free of interdependencies, in the sense that they do not involve changing the code.

Given this discussion, one might speculate that overall, in OSS projects, low post-release defect density and high productivity stem from effective use of the open source community for the low-interdependence bug finding and fixing tasks. The fact that Mozilla was apparently able to achieve defect density levels like Apache's argues that even when an open source effort maintains much of the machinery of commercial development (including elements of planning, documenting the process and the product, explicit code ownership, inspections, and testing), there is substantial potential benefit. In particular, defect density and productivity both seem to benefit from recruiting an open source community of testers and bug fixers. Speed, on the other hand, seems to require highly modularized software and small highly-capable core teams and the informal style of coordination this permits.

Interestingly, the particular way that the core team in Apache (and, we assume, many other OSS projects) is formed may be another of the keys to their success. Core members must be persistent and very capable to achieve core status. They are also free, while they are earning their core status, to work on any task they choose. Presumably they will try to choose something that is both badly needed and where they have some specific interest. While working in this area, they must demonstrate a high level of capability, and they must also convince the existing core team that they would make a responsible, productive colleague. This is in contrast to most commercial development, where assignments are given out that may or may not correspond to a developer's interests or perceptions of what is needed.

In contrast to Apache, the Mozilla project began as a commercial endeavor, and was only later morphed into an open source approach. Mozilla provides us with a hybrid example that illustrates some of the possible properties of open source development techniques used in a commercial context.

The Mozilla approach has some, but not all, of the Apache-style OSS benefits. The open source community has taken over a significant portion of the bug finding and fixing, as in Apache, helping with these low-interdependency tasks. However, the Mozilla modules are not as independent from one another as the Apache server is from its ancillary projects. Because of the interdependence among modules, considerable effort (i.e., inspections) needs to be spent in order to ensure that the interdependencies do not cause problems. In addition, the modules are too large for a team of 10-15 to do 80% of the work in the desired time. Therefore, the relatively free-wheeling Apache style of communication and implicit coordination is likely not feasible. The larger Mozilla core teams must have more formal means of coordinating their work, which in their case means a single module owner who must approve all changes to the module. These characteristics produce high productivity and low defect density, much like Apache, but relatively long development intervals.

The relatively high level of module interdependence may be a result of many factors. For example, the commercial legacy distinguishes Mozilla from Apache and many other purely open source projects. One might speculate that in commercial development, feature content is driven by market demands, and for many applications (such as browsers) the market generates great pressure for feature richness. When combined with extreme schedule pressure, it is not unreasonable to expect that the code complexity will be high and that modularity may suffer. This sort of legacy may well contribute to the difficulty of coordinating Mozilla and other commercial-legacy hybrid projects.

It may be possible to avoid this problem under various circumstances, e.g.,

- new hybrid projects that are set up like OSS projects, with small teams owning well-separated modules,
- projects with OSS legacy code, and
- projects with a commercial legacy, but where modules are parsed in a way that minimizes module-spanning changes (see [4] for a technique that accomplishes this).

We believe that for some kinds of software, in particular those where developers are also highly knowledgeable users, it would be worth experimenting, in a commercial environment, with OSS-style "open" work assignments. This approach implicitly allows new features to be chosen by the developers/users rather than a marketing or product management organization.

It is tempting to suggest that commercial and OSS practices might be fruitfully hybridized in a number of ways. For example, it might prove very attractive to commercial developers to use the OSS style project structure. In such an arrangement, there is a core team of recognized experts, who alone have the power to commit code to an official release, and a much larger group who contribute voluntarily in various ways, and who may prove themselves diligent and skillful enough to be added to the core. Everyone, under this type of project management, is self-determining. The core members can commit code where they choose, the peripheral members submit changes of any sort they choose. These decisions appear to be guided only by a common desire to see the product developed successfully, to contribute in meaningful ways, and to be seen as an important contributor.

While we are certain that this suggestion will be met with healthy skepticism, we see no inherent reason why commercial developments could not operate in a similar manner, subject of course to restrictions on size, and the necessity that developers must be users. Assuming that this arrangement would work in a commercial setting, there could be tremendous benefits to pairing the high motivation, low pre-system test defect rates, and fast response of OSS with a more commercially-oriented system test capability. Such cross-fertilization might pave the way to a true revolution in software development.

Two areas that seem particularly promising for the introduction of OSS techniques in the commercial world are tools and platforms. Developers generally create a variety of in-house tools for their own use, or for the use of their work groups. They are generally fairly small, and the developers are obviously users. Assuming that there are tools with sufficiently general utility, they could provide a natural place for trying out OSS techniques with relatively low risk.

A second area is platforms, i.e., software that provides a more specialized layer on top of an operating system, on which a number of distinct products can be built. Members of project teams might be permitted (or persuaded) to spend some portion of time on platform development, possibly working their way into the core team. Again, product developers are users of the platform, and no doubt would like to see specific improvements made to accommodate their product.

REFERENCES

[1] GRINTER, R. E., HERBSLEB, J. D., AND PERRY, D. E. 1999. The Geography of Coordination: Dealing with Distance in R&D Work. In GROUP '99, Phoenix, AZ

[2] HERBSLEB J. D., AND GRINTER, R. E. 1999. Splitting the Organization and Integrating the Code: Conway's Law Revisited. In 21st International Conference on Software Engineering (ICSE 99), Los Angeles, CA.

[3] MOCKUS, A., FIELDING, R., AND HERBSLEB, J. 1999. A case study of open source development: The apache server. In 22nd International Conference on Software Engineering, pages 263-272, Limerick, Ireland, June 4-11 2000.

[4] MOCKUS, A., AND WEISS, D. M. 2001. Globalization by Chunking: A Quantitative Approach, IEEE Software, vol. 18, No 2, January - March, 30-37.