Weighing the value of changeability in Open Source Software

Mattia Monga, Andrea Trentini
D.I.Co. - Università degli Studi di Milano
Via Comelico 39, 20135 Milano, Italy
(monga|trentini)@dico.unimi.it

April 14, 2006

Abstract

Software value assessment is not an easy task. Many techniques have been proposed in the past, qualitative and quantitative, based on human evaluation or on measureable metrics. This paper proposes a quantitative technique based on the concept that "modularization is value", applied to the Debian package database. This (software implemented) technique defines two metrics: *change cost* (as a function of the dependencies among packages) and *integration cost* (as a function of the number of maintainers involved). We applied our technique to sets of packages and we show and comment the results we obtained.

1 Introduction

A big issue in Software Engineering is the task of evaluating the "quality" of a software system. Traditional SE techniques span from the use of metrics [8, 7, 2, 4] to the measurement of defects by testing a system at runtime [1, 10, 5].

Another important issue to be assessed is the "value" of a piece of software, and, in particular, its "relative value" (e.g., relative to a specific user).

The quality of a software system influences its "value": if a low quality (e.g., many bugs/defects) means always low value (no matter the purpose of the system), high quality does not imply high value since the definition of "value" should take into account functionality (in a broader sense) and subjective usefulness. That is, the measurement of software value can also be done by evaluating the "suitability to task" for a specific user (or group of).

One of the techniques proposed to evaluate the suitability of open source software is BRR (Business Readiness Rating^{TM} [9]). BRR is a community initiative (sponsored by Carnegie Mellon West Center for Open Source Investigation, O'Reilly CodeZoo, SpikeSource and Intel) to supply a standardized method to decide if some specific Open Source software application is ready (mature enough) to be used in a business context. BRR defines a set of semi-quantitative metrics (Functionality, Usability, Quality, Security, Performance, Scalability, Architecture, Support, Documentation, Adoption, Community, Professionalism) and gives guidelines to evaluate and to normalize (by weighing) them for the computation of a "readiness-for-business score".

In our view, the main disadvantage of BRR is that the defined metrics are not purely quantitative and they cannot be *measured on the code,* although it is exactly the availability of the code the key difference between open source and closed source products. The BRR guidelines offer a standard method to frame votes in a repeatable schema. However, a way to (somewhat) measure the value of a software product through a purely quantitative (and possibly automatic) manner is also needed.

In the following we propose a quantitative technique to evaluate the effort needed to introduce a change in a set of *Debian packages* [3]. By exploiting the Debian APT database (a cache of all available packages), which is very complete in terms of meta-information on dependencies between packages, we calculate the impact of introducing a change and integrating them in the system. Our metrics are in part derived from [6].

2 Proposed approach

The very essence of open source software (OSS) is that it is possible to change its code. Closed source products can be customized along the lines their creators have provided by configuration means. OSS, instead, can be modified even in order to achieve unanticipated features. Thus, if in closed source a key property is *flexibility,* i.e. how easy is to adapt a product to a specific task, the value of OSS depends also on its *modifiability,* i.e., how easy is to modify the code. Moreover, since modifications are often carried on concurrently with the *upstream* development, it is also important to assess how expensive is to keep coherence between both development trunks (upstream and downstream: possibly eventually merging them).

In order to measure the changeability of a piece of OSS, we consider it as composed by *a number of units* that can be the target of a change. These can be modules, in the sense of an encapsulated functionality with well defined boundaries, files, packages, or even arbitrary sections of code. What is important in our approach to changeability evaluation is that should be possible to know the *dependencies* among different units. A unit A depends on unit B if a change in the semantics of B changes the semantics of A.

The knowledge about dependencies among units can be used to measure the coupling of the whole system. Following [6], we call *change cost* the degree to which a change to any single unit causes a (potential) change to other units in the system, either directly or indirectly through a chain of dependencies.

Moreover, when a change is introduced in an open source system, we would like to assess how easy would be to integrate it in the upstream development or, conversely, how easy will be to merge further upstream releases with the downstream change. We call this *coordination cost* and we suggest that it is a function of the number of principals involved in the development and the average change cost of the system.

2.1 A case study with Debian packages

As a case study for our changeability assessment we choose the Debian packaging system. We considered a set of Debian packages and their dependencies. In fact, every Debian system maintains a database of available and installed packages. Packages are produced by Debian maintainers by augmenting pieces of OSS with control information that is used to install them coherently on users machines. Debian adopts a "micro-packaging" policy, i.e., whenever is possible maintainers try to break a product in several small packages in order to foster reuse of libraries and shared data.

Control information includes knowledge about installation dependencies. A package A may declare a dependency on some other package B by specifying a `Depends`, `Pre-Depends`, `Recommends`, `Suggests`, `Enhances` or `Conflicts` relationship. Dependencies can be analyzed by standard Debian tools: only dependencies explicitly declared by package maintainers are recorded in control information, however they are normally quite precise since the Debian developer toolkit provide several tools to discover and track them. In our preliminary study we have taken into account only `Depends` constraints.

2.1.1 Change cost computation

Given a set of packages, we compute the change cost from the dependency matrix. A dependency matrix is a binary square matrix in which the element $a_{i,j}$ is 1 if and only if the package i depends on the package j, 0 otherwise.

Indirect dependencies can be computed by matrix multiplication. $M \times M = M^2$ contains the dependencies obtained in 2 steps. In fact:

$(M^2)_{i,j} = \sum_{r=1}^{n} (M)_{i,r}(M)_{r,i}$

thus $(M^2)_{i,j} > 0$ if and only if r exists such that i depends on r and r depends on j.

M^0 is the identity matrix, for each i, $(M^0)_{i,i} = 1$, and it can be interpreted as the fact that each package depends on itself.

If n is the number of packages in the set, the transitive closure of dependencies T can then be computed by the formula

$$T = \sum_{k=0}^{n} (\hat{M}^k) \tag{1}$$

where $\hat{M}^k$ is the matrix M^k where any $(M^k)_{i,j} > 1$ was substituted by 1. Limiting the sum to n avoids

the problem of cyclic dependencies that might be present in the package database.

The density of ones in T is the measure of change costs we considered. In fact, the sum of values of every element in a column j of T gives how many other packages are potentially affected by a change in j. The average change cost χ of the set of package is given by the formula

$$\chi = \frac{\sum_{i=0,j=0}^{n} (\hat{T})_{i,j}}{n^2} \tag{2}$$

where $\hat{T}$ is the matrix T where any $(T)_{i,j} > 1$ was substituted by 1

The maximum value for χ is 1, meaning that every change impacts on everything. The minimum value is $1/n$, when each package depends only on itself.

2.1.2 Integration cost computation

Integration cost is a measure of the effort needed when a change is introduced in the system and we want to merge that with the upstream development. Intuitively, this cost increases with the number of people which are affected by the change, since the change merge has to be ideally coordinated among all of them. Thus, if the change cost χ gives the average percent ratio of units affected by a unit change, we consider the coordination cost κ given by

$$\kappa = \chi \cdot p \tag{3}$$

where p is the number of principals involved in the development of the system. Given a set of packages, p is the number of Debian maintainers who are responsible for all of them.

3 Experimental data

We built a Python script to compute change and integration costs on arbitrary sets of Debian packages, leveraging on the APT cache database. The tool is available on request. We ran our tool over several subsystems with the same functional goal: in other words we tried to get cost figures about comparable applications.

Thus, we compared:

- two implementations of the X Window System, the old monolithic XFree86 and the new modular one, XOrg;
- two graphical desktop managers, Gnome and KDE;
- two interpreted language environments, namely Python and Perl;
- five mail servers;

Pkg set	**# of pkgs**	**Change cost**	**Integration cost**
xorg	66	0.028696	0.057392
xfree86	3	0.444444	0.444444
kde	359	0.005912	0.384308
gnome	380	0.006053	1.004737
perl	1068	0.006667	1.526835
python	941	0.004118	0.650571
postfix	9	0.185185	0.740741
qmail	1	1	1
cyrus	25	0.060800	0.304000
sendmail	4	0.562500	0.562500
courier	25	0.139200	0.278400

Table 1: Change and integration costs for several package sets

Table 1 shows the results of the computation of the *change cost* and the *integration cost*. Here in the table we have four groups of package sets:

Some interesting remarks can be made over these results:

- Consistently with our intuition a modularized product gives a lower change cost with respect to a monolithic one (see XFree86 vs. XOrg);
- A high number of packages in the set lowers the change cost since most packages have a small number of dependencies, in the range 0-100 (maybe less). The dependency matrix is quite sparse;
- The KDE development team is smaller than the Gnome team. In fact, the integration cost is far lower meaning that a change has to be negotiated with less people;
- When comparing Cyrus and Courier mail servers (same amount of packages): Cyrus has lower change cost but a higher integration cost. It seems that Cyrus is better structured (modularized) but since much more developers are

involved, integrating a change in it could be hard, thus "wasting" part of the modularization advantage.

4 Conclusions

In this paper we proposed two measures for weighing the value of changeability in Open Source Software products. We applied our metrics to some complex subsystems of the Debian GNU/Linux distribution. Since open source is mainly about "getting and modifying the code" we wanted to have some systematic way to assess the cost of a change in the code base. We considered two different costs: the cost of actually performing the change (change cost) and the cost of integrating the change in the open source system. Our preliminary results are consistent with the well accepted software engineering principle that modularity fosters changes. Integration cost is even more difficult to assess. We choose to evaluate the development "entropy" that one has to manage for integrating a change with the number of people involved in the work. In fact, lot of other factors are probably relevant, as, for example, how open to external contributions a development community is. We are working on a refinement of our metrics and we are planning to assess them on a more extensive set of examples.

References

[1] W. Richards Adrion, Martha A. Branstad, and John C. Cherniavsky. Validation, verification, and testing of computer software. *ACM Comput. Surv.*, 14(2):159–192, 1982.

[2] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 592–605, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[3] Debian. http://www.debian.org, 2006.

[4] Norman E. Fenton and Martin Neil. Software metrics: roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 357–370, New York, NY, USA, 2000. ACM Press.

[5] D. Gelperin and B. Hetzel. The growth of software testing. *Commun. ACM*, 31(6):687–695, 1988.

[6] Alan MacCormack, John Rusnak, and Carliss Y. Baldwin. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. In *Management Science (forthcoming)*.

[7] Nachiappan Nagappan, Laurie Williams, Mladen Vouk, and Jason Osborne. Early estimation of software quality using in-process testing metrics: a controlled case study. In *3-WoSQ: Proceedings of the third workshop on Software quality*, pages 1–7, New York, NY, USA, 2005. ACM Press.

[8] Hideto Ogasawara, Atsushi Yamada, and Michiko Kojo. Experiences of software quality management using metrics through the lifecycle. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 179–188, Washington, DC, USA, 1996. IEEE Computer Society.

[9] Business Readiness Rating. Rfc 1. `http://www.openbrr.org/docs/BRR_whitepaper_2005RFC1.pdf`, 2005.

[10] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.