Adapting the “Staged Model for Software Evolution” to Free/Libre/Open Source Software

Andrea Capiluppi
Centre of Research on Open Source Software
University of Lincoln, UK
acapiluppi@lincoln.ac.uk

Jesús M. González-Barahona, Israel Herraiz, Gregorio Robles
GsyC/LibreSoft
Universidad Rey Juan Carlos – Madrid, Spain
{jgb,herraiz,robles}@gsyc.escet.urjc.es

ABSTRACT

Research into traditional software evolution has been tackled from two broad perspectives: that focused on the *how*, which looks at the processes, methods and techniques to implement and evolve software; and that focused on the *what/why* perspective, aiming at achieving an understanding of the drivers and general characteristics of the software evolution phenomenon.

The two perspectives are related in various ways: the study of the what/why is for instance essential to achieve an appropriate management of software engineering activities, and to guide innovation in processes, methods and tools, that is, the how. The output of the what/why studies is exemplified by empirical hypotheses, such as the staged model of software evolution,.

This paper focuses on the commonalities and differences between the evolution and patterns in the lifecycles of traditional commercial systems and free/libre/open source software (FLOSS) systems. The existing staged model for software evolution is therefore revised for its applicability on FLOSS systems.

1. INTRODUCTION

The phenomenon of software evolution has been described in the literature (e.g., [10, 11]), with several models of different nature ([1, 2, 12, 16]) being proposed to understand and explain the empirical observations. Some of these models purport to be universally applicable to all software development processes. However, the models in the literature were built mainly observing software developed using a traditional centrally-managed waterfall development process, or one of its variants [20].

Research in this area has been approached from two different perspectives. One considers the processes, methods and techniques to implement and evolve software (the *how*). The other applies systematic observation of empirical data to achieve an understanding of the causes and general characteristics of the phenomenon (the *what/why*). Both perspectives are related: the study of the what/why is important in order to achieve and appropriate plan, manage and control the various software engineering activities; and to guide the development of new processes, methods and tools, that is, to guide the how.

The link between the how and the what/why perspectives is illustrated in [13], where several guidelines are derived from Lehman’s laws of software evolution. The output of the what/why study is exemplified by empirical generalizations such as Rajlich and Bennett’s model of the lifecycle [16] and Lehman’s laws of software evolution ([11, 15]).

In this context, the present paper expands and refines the empirical hypothesis presented in the staged model of software evolution [16] so that it can be applied to FLOSS projects. For this, we compare and contrast the existing empirical knowledge (e.g. as derived from studies of proprietary systems evolved under traditional processes [9]) with the emergent FLOSS paradigm. This revision will help FLOSS developers and practitioners to characterize any FLOSS system, in terms of which phase it is currently undergoing, or which phase it will more likely move to.

2. THE STAGED MODEL

The *staged model for software evolution* represents the software lifecycle as a sequence of steps [16]. Based on the traditional commercial projects, the core idea of the model is that software systems evolve through distinct stages: - Initial development, or *alpha stage*, includes all the phases (design, first coding, testing) achieved before the first running version of the system. In this stage, usually no releases are made public to the users.

- Evolutionary pressures enhance the system with new features and capabilities in the phase of the *evolution changes*: binary releases and individual patches are made available to the users, and feedback is gathered to further enhance the system.

- As long as the profitability of either new enhancements or changes to the existing code base is overcome by the costs of such modifications, the *servicing phase* is reached. The system is considered mature, changes are added to the code base, but no further enhancements (apart from patches) are provided to the end users.

- When the service is discontinued and no more code patches are released, the stage of *phase-out* is meant to declare the system’s end. This can be associated with the presence of a new enhanced system substituting the old one.

- The old system serves as a basis for the new one and then it is *closed down.*

IWPSE’07 September 3-4, 2007, Dubrovnik, Croatia
Copyright 2007 ACM 978-1-59593-722-3/07/09 ...$5.00.

3. FLOSS STAGED EVOLUTION MODEL

The previous section introduced the details of the staged model for software evolution which successfully models many traditional commercial systems. This takes us to the main research question of this paper: is this model also suitable for FLOSS systems? Building upon the results obtained for FLOSS systems by ours and others case studies, it is possible to analyze each of the phases of the cited model, and observe when and how differences and commonalities arise.

3.1 Initial development

Traditional commercial systems are typically built by a fixed amount of designers, developers and testers [17]. Also, software is usually published (as executables) only after the first release is deemed as "correctly running". The recipients of this release are the end users. It is a rare event taht source code access is granted too: we are aware of just one specific case in which a commercial software house, using an agile development process, gives the possibility for users to download the application from the public versioning system repository [3].

FLOSS systems, on the other hand, typically start with a small amount of early developers, and eventually new developers join after a certain initiation period [21]. FLOSS systems, instead, may be released (in binaries as well in source code) well before they are complete or working. Recipients are not only end-users, but also any other developer: read-only access to the versioning system is given to anybody, which leads to the already mentioned *continuous release* even before the first official release [17]. In the following, results from empirical case studies are reported to justify the mentioned differences.

3.1.1 Case studies

Successful FLOSS projects have been studied and characterized in the past [1, 8, 18], but empirical evidence on their behavior in the initial development, and the transition to a larger "evolution" phase has not been proven yet.

The initial development phase in FLOSS projects has been characterized as a cathedral-driven software process, as contributions from external developers are not yet encouraged [4, 21]. The process is hence controlled, the infrastructure for the project is not always in place, and the feedback from end-users is limited. Major differences appear when a FLOSS project either never leaves this initial stage, as documented for a large majority of the projects hosted on SourceForge [6]; or when it leverages a "bazaar", i.e. a large and increasing amount of developers.

Empirical studies have started to characterize the initial development of a FLOSS project [4]. A closed process, performed by a small group of FLOSS developers, has some commonalities with traditional software development: one observed system (Arla, a Network File System), was shown to have remained, through its lifecycle, an effort of a small team [17], or a "cathedral". Also the output was shown to be following the same constant trend. These results were not attributed to an unsuccessful FLOSS project, but interpreted as a potentially missed opportunity to establish a thriving community around a project.

Specific actions from the core developers (or lone project author) have also been identified when a FLOSS project was to leave this initial stage. It has been shown that new developers in both the analysed systems prefer to work on newly added modules, rather than older ones: therefore, it was argued that core developers should create new avenues of development to let new developers join in. Further analyzing this system, a decreasing trend of new module creation was detected, which prevented new developers to join in [4].

In another case study (Wine, a free implementation of Windows on Unix), a growing trend of developers and output produced was observed. Also, the amount of new modules inserted by the core developers follows a similar growing trend. This helped with the recruitment of new developers and to leave the initial stage for a "bazaar" stage [17].

This first difference between traditional and FLOSS systems is annotated in the revised model displayed in Figure 3: the box containing the "initial development phase" is highlighted, as it could be the only phase available in the evolution of a FLOSS system. Also, in the same phase, a different handling of the versioning system is achieved.

3.2 Evolution changes

Several releases are observed both in traditional commercial and FLOSS systems. The staged model, as proposed, also incorporated the possibility of evolution through several branches of releases [16]. In traditional commercial systems, most of the changes are distributed and applied as patches on the existing code base. New versions of the software systems are distributed regularly, albeit a higher frequency is perceived as an instability factor. Feedback is provided by users in the form of requests for change or bug signaling, and collected as a set of new requirements for new releases, or in intermediate code patches.

In FLOSS systems, new releases of systems and patches are available more often, and this is usually perceived by FLOSS developers as a vitality factor [7, 17]. Although traditionally many FLOSS projects published a new release "once it is ready", in recent times several FLOSS projects have moved to a time-based release planning, offering a new stable version for end-users on a periodic basis (for Ubuntu and GNOME, for instance, every six months, [14]). Feedback is provided by users in the same forms as in commercial systems, but also under the form of code patches that users write themselves, and which possibly will be incorporated into new releases of the system.

The loop of evolution changes presented in Figure 3 may be accomplished through many years. Both traditional commercial and FLOSS systems have shown the characteristics of long-lived software. For instance, operating systems like OS360, the various flavors of UNIX, or the Microsoft Windows, as well as the FLOSS Linux kernel, FreeBSD or OpenBSD, have been successfully evolving for decades.

It is noticeable that while the evolution loop can be found both in commercial and FLOSS environments, several research papers have shown that growth dynamics in both cases differ significantly, at least in the case of large projects [8, 18]. Some of the FLOSS projects have a superlinear growth rate (for example, Linux [8]), while a majority of the large projects studied grow linearly. Both behaviors (superlinearity and linearity) seem to be in contradiction with Lehman's laws of Software Evolution that imply that size over time shows a decelerated pattern [22].

3.3 Servicing

This phase was first described for traditional commercial systems, when new functionalities are not added to the code

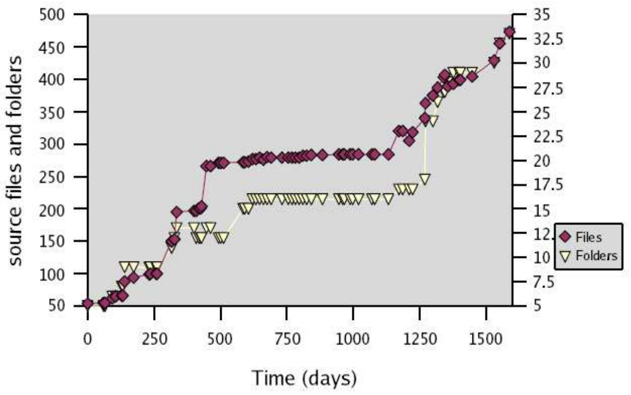

Figure 1: Growth, stabilization and further growth in the gaim system

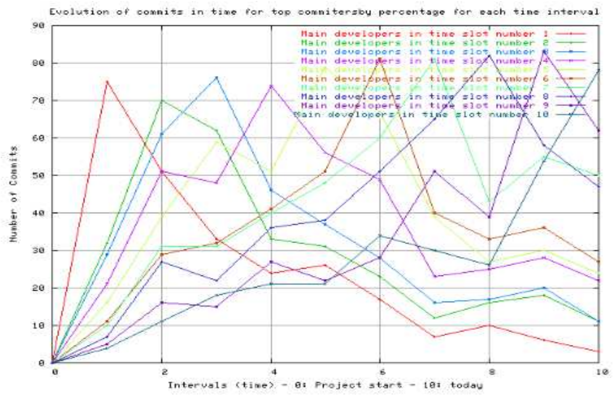

Figure 2: evolution of top most committers in one of the analysed projects

base, whilst fixes to existing features are still performed [16]. The transition from evolution to servicing is typically based on the economic profitability of the software system. When revenues from a software product are not balanced by the costs of its maintenance, the system is no longer evolved, and it may become a legacy system [5].

For FLOSS systems, on the other hand, the evolutionary behavior shows often stabilization points, where the size of the overall system does not change, albeit several releases are made available, and a long time interval is achieved. Although a servicing stage could be detected, a new evolution period is later found. Researchers have also assessed that some 20% of the overall number of projects within SourceForge are "tragedies of growth", i.e. they evolved for a period, but then fewer and fewer additions were made [6].

The presence of servicing stages has been detected in the past through an overall small increase of the code base (say, less than 10% over several releases and temporal time, [2]). It was observed, in some of the systems, a very fast increase in size, and a corresponding fast evolution, followed by a stabilization phase which lead to the complete abandonment of the project.

For some other analyzed systems, instead, (e.g. the Gaim system, 1), albeit the same initial fast growth rate, and a transition from evolution to servicing, were observed, a new period of evolution was also found. In yet other systems (the Grace system), it was possible to observe an overall growing pace even when the system was abandoned by the initial lone developer, and was handed over a new team of developers, followed by a later stabilisation period.

The dashed arc between the evolution and servicing stages of Figure3 is displaying this possibility.

3.4 Phase Out

In traditional commercial systems, the phase out of a software system happens when the software house declares that neither new functionalities, nor the fixing of existing ones will be performed. The system becomes then a legacy application.

The same behaviour is detectable in FLOSS systems [17], when development teams declare their intention not to maintain the system any more. The main difference between the traditional commercial approach and the FLOSS cases is the availability of source code. In some, specific cases new developers may take over the existing system, and with the availability of source code, bring it to a new stage of evolution. A case study was presented here to describe the possibility (the Grace system), but literature reports others [17]. The dashed line in Figure 3 displays this possibility and revises the transitions among stages.

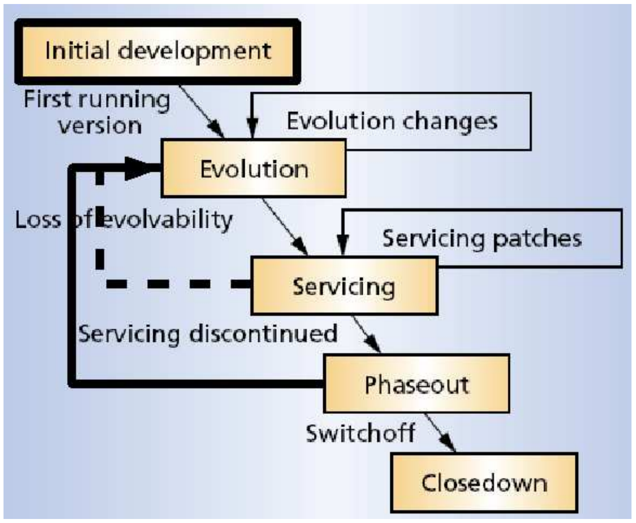

Figure 3: Staged model adapted to FLOSS systems

Many FLOSS projects have been reported to renew their core groups. For instance, in [19] three different categories of FLOSS projects were identified: code gods, generations and mixed behavior. Code gods projects are maintained by the same group of developers during the whole lifetime of the project. Generations projects exhibit a renewal in the core group of developers; the group of people that were the main developers at a early moment in the lifetime are not the main developers in posterior moments, like in Figure 2. Therefore, a generational relay has taken place in the project. Mixed projects exhibit neither a pure code god or generations profile, but a intermediate state among those two extremes.

4. CONCLUSIONS

This paper has argued that the FLOSS development cycle may be considered different from traditional commercial system. The staged model for the software evolution, as in its original form expressed in [16] model, was discussed. A general resemblance between commercial and FLOSS evolutionary behavior was recognized: initial development tend to be superlinear or at least with sustained growth (see for instance Figure 1). A stabilization point where fewer functionalities were added has been recognized in some FLOSS evolutionary behavior (central part of Figure 1). Apart from the commonalities, three points of difference were detected for enhancing the staged model.

The first is relative to availability of releases: commercial companies make software systems available to third parties only when they are running and are tested enough. On the contrary, FLOSS systems are available in versioning system repositories well before first official release, and may be downloaded at any time. The second difference is relative to the transition between the evolution stage and the servicing stage: we encountered several cases in which a new development stage was achieved after a phase without major enhancements. The third revision made to the model is a possible transition between the phases of phase out and evolution: a case was illustrated in which a new development team took over the responsibility of a project that was declared closed (Grace) . More in general, generations of developers have been identified in several FLOSS systems, where the most active developers (in terms of commits) get replaced frequently along the lifecycle of a FLOSS application. Therefore, it may be concluded that after some modifications, the original staged model for software evolution can be extended to consider the evolution of a FLOSS project.

5. REFERENCES

[1] M. Aoyama. Metrics and analysis of software architecture evolution with discontinuity. In *IWPSE '02: Proceedings of the International Workshop on Principles of Software Evolution*, pages 103–107, New York, NY, USA, 2002. ACM Press.

[2] A. Capiluppi. Models for the evolution of os projects. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 65, Washington, DC, USA, 2003. IEEE Computer Society.

[3] A. Capiluppi, J. Fernandez-Ramil, J. Higman, H. C. Sharp, and N. Smith. An empirical study of the evolution of an agile-developed software system. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 511–518, Washington, DC, USA, 2007. IEEE Computer Society.

[4] A. Capiluppi and M. Michlmayr. From the cathedral to the bazaar: An empirical study of the lifecycle of volunteer community projects. In J. Feller, B. Fitzgerald, W. Scacchi, and A. Silitti, editors, *Open Source Development, Adoption and Innovation*, pages 31–44. International Federation for Information Processing, Springer, 2007.

[5] N. Chapin, J. E. Hale, J. F., Ramil, and W.-G. Tan. Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13(1):3–30, 2001.

[6] R. English and C. Schweik. Identifying success and tragedy of floss commons: A preliminary classification of sourceforge.net projects. In *Proceedings of the 1st International Workshop on Emerging Trends in FLOSS Research and Development*, Minneapolis, MN, 2007. ICSE.

[7] D. M. German and A. Mockus. Automating the measurement of open source projects. In *Proceedings of the 3rd Workshop on Open Source Software Engineering*, Portland, Oregon, USA, 2003.

[8] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*, page 131, Washington, DC, USA, 2000. IEEE Computer Society.

[9] M. Lehman and J. Ramil. Feast: Feedback evolution and software technology.

[10] M. M. Lehman. Programs, cities, students, limits to growth? In D. Gries, editor, *Programming Methodology*, pages 42–62. 1978.

[11] M. M. Lehman and L. A. Belady, editors. *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[12] M. M. Lehman, G. Kahen, and J. F. Ramil. Behavioural modelling of long-lived evolution processes: some issues and an example. *Journal of Software Maintenance*, 14(5):335–351, 2002.

[13] M. M. Lehman and J. F. Ramil. Rules and tools for software evolution planning and management. *Ann. Softw. Eng.*, 11(1):15–44, 2001.

[14] M. Michlmayr. *Quality Improvement in Volunteer Free and Open Source Software Projects – Exploring the Impact of Release Management*. PhD thesis, University of Cambridge, UK, 2007.

[15] S. L. Pfleeger. *Software Engineering: Theory and Practice*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[16] V. T. Rajlich and K. H. Bennett. A staged model for the software life cycle. *Computer*, 33(7):66–71, 2000.

[17] E. S. Raymond. *The cathedral and the bazaar: musings on Linux and open source by an accidental revolutionary*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.

[18] G. Robles, J. J. Amor, J. M. González-Barahona, and I. Herraiz. Evolution and growth in large libre software projects. In *IWPSE*, pages 165–174. IEEE Computer Society, 2005.

[19] G. Robles and J. M. González-Barahona. Contributor turnover in libre software projects. In E. Damiani, B. Fitzgerald, W. Scacchi, M. Scotto, and G. Succi, editors, *OSS*, volume 203 of *IFIP*, pages 273–286. Springer, 2006.

[20] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[21] A. Senyard and M. Michlmayr. How to have a successful free software project. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, pages 84–91, Busan, Korea, 2004.

[22] W. M. Turski. Reference model for smooth growth of software systems. *IEEE Trans. Softw. Eng.*, 22(8):599–600, 1996.