Adapting the "Staged Model for Software Evolution" to FLOSS

Andrea Capiluppi[‡] Jesus M. Gonzalez Barahona[†] Israel Herraiz[†] Gregorio Robles[†]

‡ Department of Computing and Informatics, University of Lincoln, UK
† GsyC/LibreSoft, Universidad Rey Juan Carlos, Madrid, Spain
acapiluppi@lincoln.ac.uk, {jgb, herraiz, grex}@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., [1], [2]), with several models of different nature ([3], [4], [5], [6], [7]) 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 [21].

Research in this area has been approached from two different perspectives. One is based on considering the *how,* looking at processes, methods and techniques to implement and evolve software. The other is based on the *what/why,* applying systematic observation of empirical data to achieve an understanding of the causes and general characteristics of the phenomenon. 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 [8], where a rich set of management 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 [6] and Lehman's laws of software evolution ([9], [10]).

In this context, the present paper expands and refines the empirical hypothesis presented in the staged model of software evolution [6] 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, such as those shown in [7]) with the emergent FLOSS paradigm.

2. The staged model

The staged model for software evolution provided in [6] represents the software lifecycle as a sequence of steps. Figure 1 displays the visualization extracted from their model, as such specifically targeting the

traditional commercial projects. The basic idea 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, no releases are made public to the users.
- Evolutionary pressures tend to enhance the system with new features and capabilities in the phase of the *evolution changes*: both 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 recognizable. The system is considered mature, changes are added to the code base, but no further enhancements are provided to the end users. Individual code patches are distributed 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 is typically associated with the presence of a new enhanced system that will substitute the original one.
- The old system serves as a basis for the new one and then it is *closed down.*

The purpose of this paper is to use the knowledge accumulated in previous literature on FLOSS in order to detect similarities and differences between the traditional commercial and the FLOSS approaches regarding the stages of software evolution.

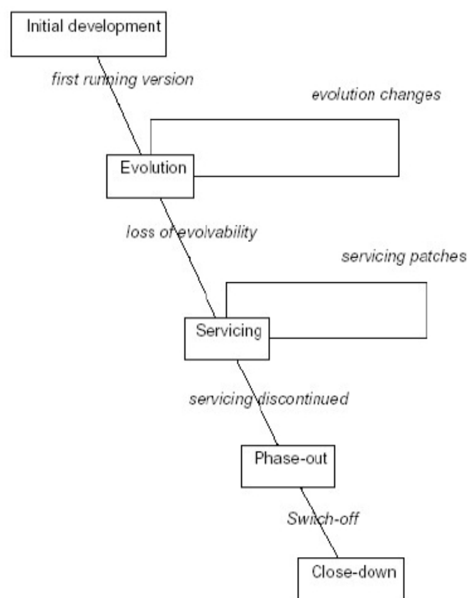

Figure 1: the staged model as first proposed in [6]

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.

In this section, three aspects of the model are revised:

- the first observed difference is based on both the description of the "initial development phase", and the definition of "available releases". As explained below, depending on the definition used for "initial phase", many FLOSS projects could be argued to never have left this stage. With respect to releases, in traditional commercial systems they have to be complete, running and authorized by the software company, while in the FLOSS world it is commonplace to allow public access to the code base in versioning system repositories, following a "permanent release" model, at least for those ready to build the product from sources, well before the official first release is published.;
- the second observed difference is related to the possibility of loops between the evolution changes and the "servicing" stage. The system goes through other recognizable phases of enhancements after a period of servicing, where no features are added. One clear case of this are the projects in which a freezing period is established before major releases. During this freezing period no new functionality is added (that is, the project is in pure servicing mode), and only after release time it goes back to evolution mode;
- the third main difference is related to FLOSS communities: new development teams may leave the ground to other developers, and therefore a FLOSS system already in the phase-out stage may experience a re-birth if a new team includes new enhancement, leading to a new evolution period.

In the following, each of the phases described in [6] are analyzed, and commonalities and/or differences are identified. Building on our previous works, empirical

evidence is given to provide a sound basis to the claims and assumptions.

3.1 Initial development

Traditional commercial systems are typically built by a fixed amount of designers, developers and testers. FLOSS systems typically start with a small amount of early developers, and eventually new developers join after a certain initiation period..

In traditional commercial systems software is usually published only after the first release is deemed as “correctly running”. FLOSS systems, however, may be released well before they are complete or working, and typically 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.

This can be true also for traditional commercial systems, but it is a rare event in that realm: 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 [23].

The initial development phase in FLOSS projects has been characterized in [11, 12] as a cathedral-driven software process, as contributions from external developers are not yet encouraged. The process is hence controlled, the infrastructure for the project is not always in place, and the feedback from end-users is limited.

3.1.1 Case studies

Successful FLOSS projects have been studied and characterized in the past, but empirical evidence on their behavior in the initial development, and the transition to a larger “evolution” phase has not been proven yet.

In [12] two case studies were selected to characterize the initial development of a FLOSS project. A closed process, performed by a small group of developers, has some commonalities with traditional software development. 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 [22]; or when it leverages a “bazaar”, i.e. a large and increasing amount of developers. Figure 2 displays the number of developers contributing to a system (the Arla Network File System), showing hot it has remained, through its lifecycle, as an effort of a small team[19]. It has been argued that this should not be interpreted as a sign of the overall failure of a FLOSS project, but as a potentially missed opportunity to establish a thriving community around a project.

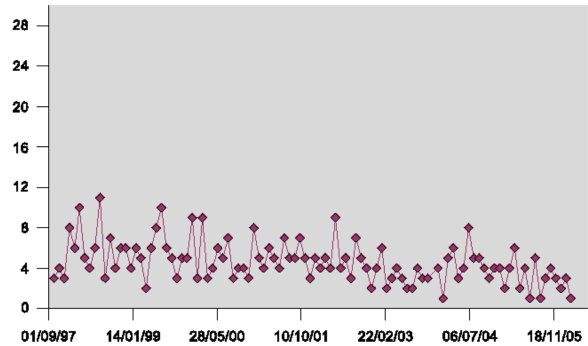

Figure 2: number of developers in a FLOSS system (Arla) not leaving its “initial development phase”

In the same study, specific actions from the core developers (or lone project author) were identified when a FLOSS project was to leave this initial stage. Since new developers prefer to work on newly added modules, rather than older ones, core developers should create new avenues of development to let new developers join in. Further analyzing the system displayed in Figure 2, a decreasing trend of new module creation was detected, which prevented new developers to join in.

Figure 3 shows the evolution of active developers in another case study (Wine, a free implementation of Windows on Unix), where a growing trend is observed. Also, as further studied, 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 [19].

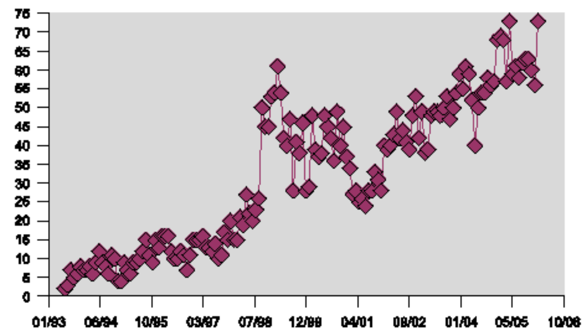

Figure 3: number of developers in a FLOSS system (Wine) that was able to leave the initial phase

This first difference between traditional commercial systems and FLOSS systems is annotated in the revised model displayed in Figure 6: 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. 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 as a vitality factor [18, 19]. 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 of the project on a periodic basis (for Ubuntu and GNOME, for instance, every six months, [20]). 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 1 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 [13, 14, 15]. Some of the FLOSS projects have a superlinear growth rate (for example, Linux), 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 [16].

3.3 Servicing

This phase was first described for traditional commercial systems, when new functionalities are not added to the code base, whilst fixes to existing features are still performed [6]. 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 [17].

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.

This behaviour can be matched with the release schedule of a typical FLOSS project. Some time before a new version is released, it is usual to "freeze" the current code. This stage can be qualified as a *servicing stage*. Right after the new release is made, changes are made to increase the functionality of the project. This could be qualified as a new *evolution stage.* Therefore, these stabilization points that use to happen in the surroundings of a new release can be identified as sequences of pairs of evolution and servicing stages, that are repeated several times during the whole lifetime of the project.

3.3.1 Case Studies

In the study reported in [4], the presence of servicing stages was detected through an overall small increase of the code base (say, less than 10% over several releases and temporal time).

We 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 abandonment of the project. For the Grace system (Figure 4) all the dots represent public releases, and it is possible to observe that their overall pace is not diminished by the system entering this phase. The circled point shows when the system was abandoned by

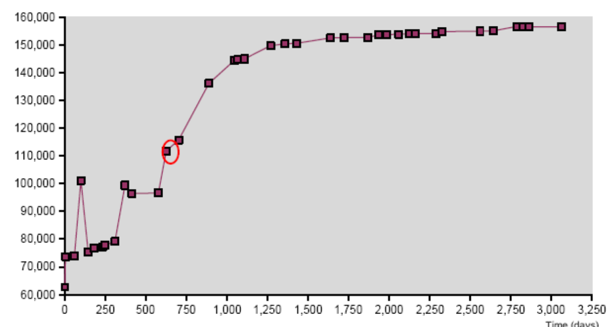

Figure 4: Stabilization point ("servicing stage") in the Grace system

the initial lone developer, and was handed over a new team of developers.

For some other analyzed systems, instead, (e.g. the Gaim system, depicted in Figure 5), albeit the same initial fast growth rate, and a transition from evolution to servicing, were observed, a new period of evolution was also found. Some other cases are summarized in Table 1. The dashed arc between the evolution and servicing stages of Figure 6 is displaying this possibility.

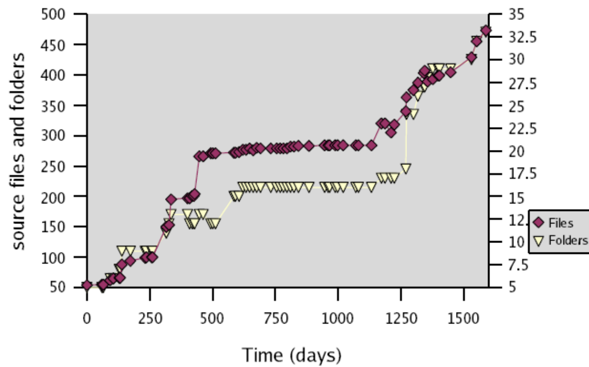

Figure 5: fast initial evolution, servicing and new evolution in the Gaim project

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.

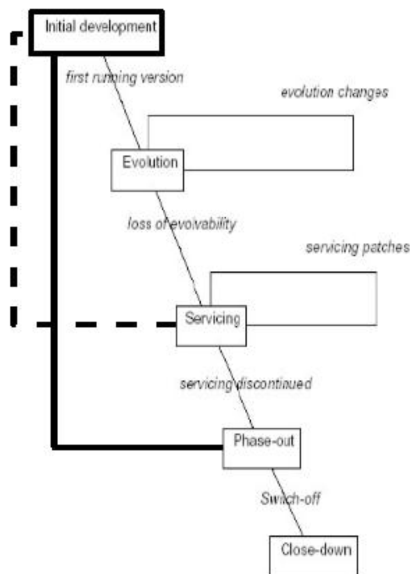

Figure 6: the staged model adapted to FLOSS systems

The same behaviour is detectable in FLOSS systems [19], 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 [19,25]. The dashed line in Figure 6 displays this possibility and revises the transitions among stages.

Many FLOSS projects have been reported to renew their core groups. For instance, in [24] 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. 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. For instance, Figure 7 shows the *generations* for the Mozilla project [23].

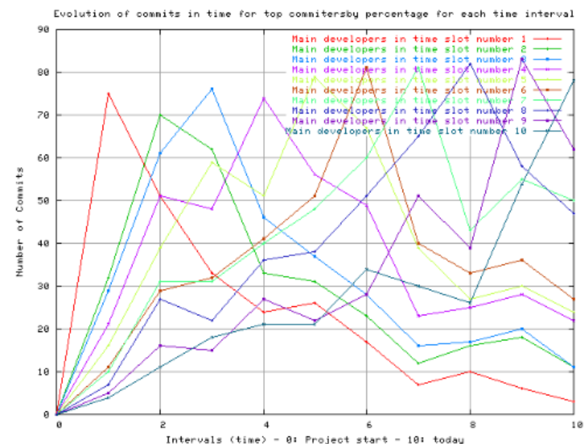

Figure 7: evolution of top most committers in the Mozilla project (from [23])

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 [6] 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 (MPlayer, Arla, Ganymede, see Table 1). A stabilization point where fewer functionalities were added has been recognized in some FLOSS evolutionary behavior (Ganymede, Grace, see Table 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 at least three cases (ARLA, Gaim, Gwydion Dylan, see Table 1) in which after a phase without major enhancements, a new development stage was achieved. The third revision made to the model is a possible transition between the phases of phase out and evolution: we illustrated a case 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* could be extended to consider the evolution of a FLOSS project.

5. References

[1] Lehman M.M., “Programs, Cities, Students, Limits to Growth?”, Inaugural Lecture, in Imperial College of Science and Technology Inaugural Lecture Series, v. 9, 1970, 1974: 211 - 229. Also in Programming Methodology, Gries D (ed.), Springer Verlag, 1978: 42 - 62.

[2] Lehman M.M., Belady L.A. (eds.), “Program Evolution - Processes of Software Change”, Academic Press, London, 538 pp.

[3] Aoyama M., “Metrics and Analysis of Software Architecture Evolution with Discontinuity”, Proc. 5th Intl. Workshop on Principles of Software Evolution, IWPSE 2002, Orlando, FL, pp. 103 – 107.

[4] Capiluppi A., “Models for the evolution of OSS projects”, Proc. of the 7th International Conference on Software Maintenance, ICSM, Amsterdam, September 22 – 26 2003, pp. 65 – 74.

[5] Lehman M.M., Kahen G. & Ramil J.F., “Behavioural Modelling of Long lived Evolution Processes– Some Issues and an Example”, Journal of Software Maintenance and Evolution, spec. issue on Separation of Concerns, vol. 14, 2002, pp. 335 – 351.

[6] K. Bennett, V. Rajlich, "Software Evolution: A Road Map," Proc. 17th International Conference on Software Maintenance (ICSM'01), 2001.

[7] FEAST, Feedback, Evolution And Software Technology, Dept. of Computing, Imperial College, http://www.doc.ic.ac.uk/~mml/feast/ (as of May. 2007).

[8] Lehman M.M. and Ramil J.F., “Rules and Tools for Software Evolution Planning and Management”, Annals of Software Engineering, vol. 11, special issue on Software Management, 2001, pp. 15 – 44.

[9] Lehman M.M. & Belady L.A, “Program Evolution - Processes of Software Change”, 1985, Academic Press, London, 538 pp.

[10] Pfleeger S.L. Software Engineering – Theory and Practice, 2nd Ed., 2001, Prentice Hall, NJ, 659 pp.

[11] Senyard A. and Michlmayr M., How to have a successful free software project, proc 11th Asia-Pacific Software Engineering Conference, pages 84-91, Busan, Korea, 2004. IEEE Computer Society.

[12] Capiluppi A. and Michlmayr M., From the Cathedral to the Bazaar: An Empirical Study of the Lifecycle of Volunteer Community Projects, proc. 3rd International Conference on Open Source Software, 2007, Limerick, Ireland.

[13] Godfrey, M., and Tu Q., “Evolution in Open Source Software: A Case Study”. Proc. of 2000 ICSM, October 11-14 2000, pp. 131 – 142.

[14] Koch S., “Evolution of Open Source Software Systems - A Large-Scale Investigation”, proc 1st International Conference on Open Source Systems, 2005, Genova, Italy, pp. 148-153.

[15] Robles G., Amor J.J., Herraiz I., Gonzalez-Barahona, J.M., “Evolution and Growth in Large Libre Software Projects”. In Proceedings of the International Workshop on Principles of Software Evolution, Sept. 2005, pp. 165 – 174.

[16] Turski W.M., “Reference Model for Smooth Growth of Software Systems”, IEEE Transactions on Software Engineering, Vol. 22, No. 8, pp. 599-600.

[17] Chapin N., Hale J.E., Khan K.M., Ramil J.F. and Tan W.G., “Types of Software Evolution and Software Maintenance”, Journal of Software Maintenance and Evolution, 13(1), January-February, pp 1–30.

[18] German D. M. and Mockus A., “Automating the measurement of open source projects”, Proc 3rd Workshop on Open Source Software Engineering, Portland, OR, USA, 2003.

[19] Raymond, E. S. 2001, “The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary”, O'Reilly & Associates, Inc.

[20] Michlmayr M., 2007, Quality Improvement in Volunteer Free and Open Source Software Projects – Exploring the Impact of Release Management, Ph.D. dissertation. University of Cambridge, UK.

[21] Royce W., “Managing the Development of Large Software Systems: Concepts and Techniques”, Proc. WESCON, IEEE Comp. Soc. Press, Los Alamitos, CA, also in Proc. ICSE'87, 30 March - 2 April, Monterey, CA.

[22] English R. and Schweik C., “Identifying Success and Tragedy of FLOSS Commons: A Preliminary Classification of Sourceforge.net Projects”, proc. 1st intl workshop on Emerging Trends in FLOSS Research and Development, 21 May 2007, Minneapolis, US.

[23] Capiluppi A, Fernandez-Ramil J., Higman J., Sharp H. C., Smith N., “An empirical study of the evolution of an agile-developed software system”, proc. 29th International Conference on Software Engineering, ICSE 2007, Minneapolis, MN.

[24] Robles G. and Gonzalez-Barahona J.M., "Contributor Turnover in Libre Software Projects", Proceedings of the IFIP 2nd International Conference on Open Source Software, June 2006, Lake Como, Italy.

[25] Behlendorf, B. “Open Source as Business Strategy” in Chris DiBona, Sam Ockman, Mark Stone (editors), *Open Sources: Voices from the Open Source Revolution*. O'Reilly. 1999.

System analysed	Main findings	Stage detected
Arla	Sustained period of growth up to day 1000 (Figure 19), delta 150%.Stabilization period up to day 1800 (800 days, relative delta 4%). Sudden growth period up to latest available release (600 days, relative delta 260%)	- sustained growth - stabilization - superlinear growth
Grace	Unconstrained growth up to day 700 (Figure 22), delta 54%: large delta sizes achieved between subsequent releases. New evolution period sustained by a new development team, up to day 1250: sublinear evolution pattern detected (550 days, delta 55%). Latest stabilization period, 2000 days, delta 5%	- chaotic growth - sustained growth - stabilization
Ganymede	Superlinear growth, up to day 250 (Figure 24), with a large delta achieved (205%); followed by a sublinear growth up day 400 (150 days, delta 16%). Missing data doesn't show evolution between day 400 and 1000. Between day 1000 and 2200, size stabilizes (1200 days, delta 8%).	- superlinear growth - sustained growth - stabilization
Gwydion Dylan	Initial stable period (previous history not available), before that, 220,000 LOCs achieved. From day 600 to present, sustained growth: 2200 days, delta 70% (Figure 28).	- initial growth period - sustained growth
Gist	Irregular growth curve up to day 900, several reduction in the global amount of lines of code. A period of growth is achieved up to day 1100, before a new shrinkage and stabilization.	Irregularities don't allow to detect clearly stages
Gaim	Large, unconstrained initial growth period (800 days, delta 1300%); further, sublinear growth period in lines of code, up to day 1400 (delta 20%). New superlinear growth between day 1400 and day 1900, with delta size 51%. Final sublinear growth up to present (Figure 34).	- superlinear growth - sublinear growth - sustained growth - sublinear growth
MPlayer	Large, unconstrained growth, between day 0 and 1500. Delta achieved 1600%	- superlinear growth
Netwib	Superlinear growth (from day 0 to 650, delta 1400%), followed by a first large reduction of size (delta -20%), and a second superlinear growth (day 700 to 1000, delta 40%); a second large reduction of LOCs (delta -25%) and a new stabilization are then observed	- superlinear growth - large negative delta - superlinear growth - stabilization

Table 1: evolutionary findings on FLOSS projects, along with the detected stages