Open source project life cycle and survival

Sami Lahtinen

June 28, 2020

Abstract

The role of open source is significant in modern computing. More and more software projects are produced under an open source license and an even greater portion of software utilizes open source code.

This significant role makes it necessary to be able to analyze the life cycles of open source projects. So that we can build our own software solutions on a reliable foundation, we must be able to identify the stages of open source projects, and the project health and ways to aid project survival.

This thesis presents a model for open source project life cycle and the differences of this model to the traditional, closed source software life cycle model. The life cycle of an open source project consists of stages, that don't fully map to the traditional understanding of software evolution, and particularly the transitions between life cycle stages differs greatly. This thesis also goes over the factors contributing to project survival, such as developer motivation, attracting new developers, support from commercial organizations and license choices. In addition to survival, the thesis also presents causes that lead to project failure and the open source projects' ability to be resurrected after project death.

ACM Computing Classification System (CCS)
CCS → Software and its engineering → Software creation and management
→ Collaboration in software development → Open source model

Contents

1 Preface

This document is a translation of my bachelor's thesis, which was originally written in Finnish for the University of Helsinki. Because of this, the language and conventions may not fully adhere to the conventions used in English computer science research and typos and grammatical errors may be plenty.

I have translated this document because I enjoyed the process of researching and writing my thesis, but presenting the results to my non-Finnish audience is quite difficult without a translation. I also believe in open research, and although my research here mainly amounts to compiling research made by others, I believe I have made a minor contribution to computer science and would like that contribution to be available to others.

I am also an open source advocate and throughout my journey of writing this thesis, I didn't find many sources that cover open source survival in a general sense.

Note, however, that the research made here is BSc-level. It has gone through evaluation and grading and received a 5/5 grade, so I believe I haven't misrepresented my sources or drawn totally incorrect and baseless conclusions. But bachelor's thesis grading is not necessarily the same as peer review. Remember to validate my research if you plan on presenting or citing it.

2 Introduction

Open source has a significant role in modern computing. We use open source software in our daily lives either directly or indirectly, particularly when we interact with the Internet. As many as 96 percent of web servers use the Linux operating system kernel [20] and over 78 percent utilize the Apache or nginx web server software [25].

Many of the open source projects are built by volunteers and are often managed a single developer [2]. Thus, as we build more and more software on open source foundations, we need to be able to ensure that these foundational projects remain functional and maintained. We need to be able to evaluate different stages of an open source project's life cycle and health. Additionally, it may be necessary to know what portion of open source projects fail and for what reasons, and how project's can avoid failures or survive in spite of them.

The research questions of this thesis are as follows:

- What does the life cycle of an open source project look like?
- Through what project properties can project health be analyzed?
- Why do projects fail and how can failures be avoided?
- How likely are project "deaths" and what portion of dying projects can be resurrected?

In chapter 2 we will define the life cycle of a software engineering project and the way it functions, and the ways in which the life cycle of an open source project differs from that of a closed source project. We will cover the different stages of an open source project's life cycle and the type of development activity associated with these stages. We will also define how an open source project transitions from one stage to another and the reasons and requirements of these transitions.

In sections 2.4 and 2.5 we will cover prior analysis of publicly available data on open source project life cycles, along with the identified difficulties in analyzing the data. We will also go through the results drawn from this data.

In chapter 3 we will focus on the health of open source projects and the project qualities that positively aid project survival. In section 3.1 we inspect the role of different kinds of developers in an open source project and what motivates these developers to participate in projects. Section 3.2 we focus on project-scale solutions, which have been identified as having a positive impact on project survival, such as technical tools and organizational structure.

In section 3.3 we cover commercial participation in open source projects. We go over the ways in which commercial organizations participate in the development of open source software and the effects of this participation on the open source projects. We will also cover different licenses used in open source software development and the effects licensing has on the commercial interest in a project and the overall health of the project.

In chpater 4 we turn our attention to open source project failure, the reasons for these failures and the ability to detect dying projects. However, we will also present the ability of open source projects to be resurrected and we will inspect how this happens in practice.

3 Open source project life cycle

In this chapter we will explore the life cycle of software development projects and particularly the life cycle of open source projects. We will cover the differences between the life cycles of open and closed source projects, the stages of an open source project's life cycle and its length and the project properties that affect life cycle length.

3.1 The life cycle of a software engineering project

Life cycles of software engineering projects have been researched since 1970s and the early life cycle models are from that era [19]. Back then it was discovered that software projects were rarely fully complete by the time they were adopted and that software development continued as the requirements placed on the software changed over the course of the project's life cycle. As a consequence the waterfall model, where a software project consists of clear and consecutive stages, didn't match the realities of software engineering. Instead software engineering projects began to be inspected from the perspective of evolution.

The theory of software evolution originates from Lehman's idea of categorizing software engineering projects into three classifications [16]. Lehman's SPE-system divides software projects to S-type programs (specified), P-type programs (problem) and E-type programs (evolution).

S-type programs are ones that can be fully specified and whose specification is not going to change. P-type programs on the other hand deal with clearly defined problems, but the solutions to these problems cannot be mathematically proved to be correct. E-type programs are made to solve problems which are either impossible to perfectly define or at least such a definition would be very difficult to make.

Because E-type programs cannot be perfectly planned ahead of time, they can only be developed by allowing evolution during the project life cycle. One of the life cycle models allowing this kind of development is a staged model, where the project is divided into 5 stages: initial development, evolution stage, maintenance stage, phase out stage and project closure [19]. In this kind of a model the evolution stage describes the phase of development, where the requirements set for the program can change and thus the program could be radically altered. In the maintenance stage following the evolution stage only minor alterations are made, which usually aim to fix errors.

This model does not describe how the evolution stage is handled in practice. Evolution could for example be handled using an iterative waterfall model, where each iteration contains all the consecutive stages of the waterfall. In modern software development this evolution would more likely to be visible in the from of agile development methods.

3.2 How do open source projects differ from traditional projects

Lehman defined a set of rules associated with the evolution of E-type programs. These laws are as follows [12]:

1. An E-type system must be continually adapted or it becomes progressively less satisfactory.
2. As an E-type system evolves, its complexity increases unless work is done to maintain or reduce it.
3. E-type system evolution processes are self-regulating with the distribution of product and process measures close to normal.
4. The average effective global activity rate in an evolving E-type system is invariant over the product's lifetime.
5. As an E-type system evolves, all associated with it, developers, sales personnel and users, for example, must maintain mastery of its content and behaviour to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves.
6. The functional content of an E-type system must be continually increased to maintain user satisfaction over its lifetime.
7. The quality of an E-type system will appear to be declining unless it is rigorously maintained and adapted to operational environment changes.
8. E-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

Lehman developed these laws by observing different, traditionally executed software engineering projects, and the laws have been determined to apply in practice in these kinds of projects [12].

In the last few decades open source software development has risen to rival traditional views on software development and the laws that apply to such development practices. Research into open source software development has revealed, for example, that Lehman's laws don't fully apply to some open source projects [12].

Some open source projects have for example been shown to break Lehman's laws related to the slowing down of development over time. From laws 2 and 4 it should follow that the growth of a project should result in dimished development efficiency. Some open source projects, such as the Linux kernel, project growth has on the other hand increased over time. This can be partially explained by Linux's modular nature, where much of the code is separated into isolated modules, which don't harm the development speed of other such modules. Open source projects also appear to be somewhat resistant to development speed decreases caused by complexity, since many such projects actively work to reduce complexity and simplify code [12]. Some projects have also been shown to reduce in size over time instead of growing. In general, project size growth seems to also fluctuate more in open source projects than in traditional software projects.

Open source software development also challenges the view on the development stages of a software engineering project. The staged model is difficult to map onto open source projects, or the transitions between stages work differently than in traditional closed source projects [5]. In open source projects the initial development phase might for example be executed in various different ways and the line between initial development and evolution phase is unclear.

The transitions between stages are also not purely linear in open source software projects, but instead open source projects can go backwards to previous stages [5]. This is largely caused by the availability of the source code and the volunteer-oriented development model. Open source projects can for example return from maintenance stage to evolution stage if the developers decide to implement new functionality. This isn't necessarily possible in traditional software projects, where projects typically enter maintenance stage because implementing new features is no longer profitable [5]. If project developers are already volunteers, their decision to implement new features does not incur additional costs. Open source code on the other hand enables other developers to resurrect projects after their original development has ended.

Because open source project life cycle so radically differs from the traditional, closed source project life cycle, an open source software specific life cycle model must be devised.

Different such models exist. One way to model open source software life cycle is to modify the traditional staged model in such ways that allow backwards transitions between stages [5]. Another way is to utilize a model based entirely on the ways in which open source projects organize themselves at different parts of their life cycle and differentiate the stages by the activity in, and organizational properties of, the project [27]. In this thesis we focus on the organizational life cycle model when inspecting the life cycle of open source projects.

This life cycle model of an open source software project consists of four stages: the initial development stage, the growth stage, the maturity stage and the decline stage [27].

In the initial development stage the software development is mostly the responsibility of one or a few developers. These developers don't yet have clearly defined roles, and instead participate in many different parts of the development process. In this stage the software is not necessarily fully usable and thus likely doesn't have many users. The goal of the initial development stage can be considered to be the production of the first working version of the software, organizing the development process and attracting new developers.

In growth stage the project begins gaining new users, and as a result, begins receiving bug reports, feature requests and possibly new source code from new users. This creates an increased managerial burden, which requires a formalization of the project structure as the original developers increasingly transition from producing source code to project management and maintenance.

In maturity stage the project reaches a point, where the role of the original developers is mainly focused on project maintenance and much of the development is delegated to other community members. Project user base continues growing but a comparatively smaller portion of these users actively participate in the development of the software.

In the decline stage the developers' and users' interest towards the project decreases. This can be caused by new competing projects or the feeling that the project doesn't need any more features. The development team shrinks to a group of core developers, who are mainly interested in maintaining the existing functionality. At the end of this stage the project faces two options: it will either be abandoned or attract new developers and users and return back to a previous stage in the life cycle.

Because two possible directions for the project's life cycle exist, the life cycle can at this point become cyclical, where it returns to growth or maturity stage and back to decline stage. Not all projects go through all of these stages. They can, for example, be abandoned before they reach the growth stage. Surveys have indicated that a significant portion of open source projects are abandoned in their initial development or growth stage [26].

3.3 Problems in analyzing open source project life cycles

Defining the life cycle of open source projects isn't completely problem-free, since the development of open source projects doesn't have centralized parties that keep track of the progress of these projects.

However, open source projects are, as the name gives away, open. Open source projects are often developed on version control platforms, like GitHub and SourceForge. Projects host repositories on these platforms, which provide access to the software's source code. These repositories also often provide means to report bugs and suggest changes to the project. Many of the studies into open source life cycle are based on the analysis of version control history, bug reports and other project metadata provided by these platforms [14].

Handling the data sets from these version control platforms proves challenging, however [14], because they contain plenty of misleading data. These data sets include, for example, large quantities of practice and hobby projects, which results in a statistical decrease in the average project activity and life expectancy. Practice and hobby projects consist of projects, where no attempt to collaborate with other developers is made and where the aim is usually learning new technologies. This leads to these projects dying quickly after the developer's interest in the topic disappears or the developer reaches the desired level of skill in the topic.

All repositories on version control platforms also don't belong in the same category. Some repositories are not related to software development, but are instead used for data storage, writing books or hosting websites. In addition some projects have multiple clones, since open source collaboration is often by creating developer-specific branches of the software and then bringing the changes made from that branch to the original repository. As a result, there can be multiple repositories associated with a single project and most of these repositories are no longer active after changes have been merged into the main repository. It is not entirely clear how these project clones should be dealt with in research.

In some studies [2] the clones of a repository are excluded from the study and a project is assumed to be dead if its original repository is determined to be inactive. This doesn't necessarily reflect reality, however, since in some cases a project's development may have simply shifted from one repository to another. This could result in a project being declared dead even if its development is still ongoing.

When projects are studied based on their version control history and metadata, only the public activity in the projects can be analyzed [14]. All unpublished development is totally invisible. This means that projects, where changes are published irregularly can appear dead even if it's still maintained. It is, however, difficult to assess how much of such unpublished work happens in open source projects. A general view in open source software development is that changes should be released early and released often [22], so it is possible that the amount of unpublished work is not significant in open source software development.

3.4 Life cycle length and its analysis

The life cycle of projects varies significantly. There are some very long-lived open source projects, which have yet to traverse all the stages of the life cycle model. When looking at the bigger picture, however, we can determine that the average life span of an open source project can be measured in weeks [17]. The problem is that this is based on a data set where short lived hobby projects were not ruled out.

More useful information about the life span of projects can be gained by inspecting popular projects, which probably rules out any such hobby projects. In a study where the data was filtered in this manner, it was revealed that project abandonment typically occurs within the first two years since the creation of the project [2]. It is worth noting that the portion of abandoned projects among popular projects was quite low. This result still gives us a more useful view into the life span of open source projects than unfiltered data.

The analysis of the unfiltered SourceForge data does indicate some project properties that correlate with the project life span [17]. For example, the programming languages used in the project affect the life span, and the study concluded that popular programming languages could result in projects being completed earlier and thus having shorter life spans. This conclusion was based on the availability of tools and libraries for popular languages, which allows projects to be developed quicker.

Another property correlating with a longer life span was the project size. Larger and more complex projects appear to live longer than simpler projects. One direct explanation for this correlation is that developing complex projects simply takes a longer time, so complex projects likely end up being more long lived. Additionally project complexity rules out smaller and shorter-lived hobby projects. Complex projects also likely solve complex problems so they are less likely to face competition from other projects.

The number of core developers in open source projects also correlated with life span in such a way, that under 20 core developers yielded an shorter average life span but in projects over 20 core developers no significant connection between number of core developers and life span was detected [17]. This means that after 20 core developers, project life span has diminishing returns. Instead the quality of these developers was found to have a significant effect. The quality of developers was assessed by the amount of fame all the developer's projects had gained in total. The study stated based on this, that developers who had previously developed popular software in the past were likelier to develop projects with a long life span.

4 How open source projects survive

In this chapter we will cover some of the basic pillars of open source project survival. We will also inspect the project properties and organizational structures that aid project survival.

4.1 Developers are the central to the survival of open source projects

Developers form the basis of the existence of open source projects. These developers can be categorized in a few different ways.

Firstly, developers can be divided into volunteers and paid developers [1]. Additionally developers can be classified by the level of their contributions. At the center of the project's development are the core developers, whose contributions are significant and who are often involved in organizing the project [15]. Some project participants are peripheral developers, who are active in the project only for a short time and contribute less. A further sub-category of peripheral developers are the one-time contributors who only make individual contributions to a given project.

Particularly the core developers are necessary to open source projects, and over time new core developers are needed to replace the ones that eventually leave the project. This creates a further need to study what motivates developers to participate in open source software development.

Developer motivations vary to some degree, particularly depending on the position of that developer in the project [15]. As an example, core developers may find participation in the project to be motivating on its own and they may have an underlying altruistic intent to produce useful code for others [15, 2]. One-time contributors and peripheral developers on the other hand are more focused on functionality benefiting themselves when making contributions. For them, participation in the project may aim to fix bugs they have detected while using the software or to improve or create features that serve their use cases.

A common motivation, however, is that the developer is already a user of the software they contribute towards [2]. This isn't surprising on its own, since projects get new developers primarily from their user base [27].

In interviews with developers contributing to projects that were previously abandoned, developers generally noted that they were aware of the state of the project prior to their participation, but this wasn't a significant factor in encouraging participation [2]. This means that developers don't usually contribute to a project purely because that project was abandoned by its previous developers. Instead the more general altruistic desire to contribute to open source software development was more prevalent among developers.

4.2 What aids or hinders project survival

In addition to developer motivations, there are other aspects that developers have noted as either aiding or hindering participation in open source projects. A friendly community and working communication channels to project maintainers have been mentioned as important components that aid participation [8]. Many technical aspects, such as availability of documentation, unit tests [8] and continuous integration [2] have also been viewed as having a positive effect on project participation. These technical measures lower the barrier to entry and provide contributors with a safety net, which prevents developers from accidentally creating regressions in the software.

Things harmful to project participation include things like the project maintainers' lack of time [8], which ties into communication channels with maintainers, and the general lack of time and experience on the part of developers [2]. In some cases new developers also lacked proper access to the source code repositories of the project, making publishing changes difficult [2]. Developers have also noted that participation in complex projects is hard [8]. This complexity can be related to the project's functionality and the problems it aims to solve, but similarly poorly selected or implemented software architecture and low-quality code can make negatively affect participatio [8].

In the longer term survival of a project, the project's popularity also plays a significant role. Even though developers can be viewed as the foundation of the existence of open source software, these developers don't appear from nowhere. As developer motivations make clear, many developers used to be users of the projects they contribute to. Open source projects thus need to reach a level of popularity to ensure the stream of new developers doesn't dry up.

GitHub provides a measure for project popularity in the from of stars. A registered GitHub user can give a repository a star, if they feel that repository to be important to them. By analyzing GitHub projects and the number of stars they have gained, we can try to find what properties drive project popularity [4]. For example, the programming language used and the project field have shown to affect project popularity. Particularly various software engineering trends tie into this popularity, as for example software related to web development is popular at the time of writing.

Popularity can also be affected by project releases. Projects have been found to get a boost in stars during the week after publishing new versions. The overall effect on project popularity is relatively small though, so project popularity cannot be thought of as being entirely determined by the frequency of releases. The effect is measurable though, which supports the “release early, release often” strategy [6].

4.3 Commercial participation in open source projects

Even though open source makes possible a software development model based on volunteering, not all open source software development is done by volunteers. Open source is also being used by commercial organizations and these organizations also participate in producing open source software.

The participation of commercial organizations in open source projects has a stabilizing effect on project survival. If a project is a critical component for the profitability of the commercial organization, the organization is probably willing to put their employees to work on the project or sponsor the project financially. This means that the slower process of attracting voluntary developers can be skipped by directly buying developer resources. Commercial sponsors also positively impact project popularity among users, and thus also developers [23].

The level of participation varies between organizations, however. Commercial organizations have been found to participate in open source projects either in a parasitic, symbiotically or commensally [11]. This classification is based on the how the open source project and the commercial entity benefit from the relationship. In a parasitic relationship, the commercial organization takes advantage of the project in ways that can be harmful to the project itself. In commensal relationships the commercial organization can benefit from the project without significantly advancing or hindering the projects development. This kind of a commensal relationship could for example be one, where a commercial organization uses an open source project without making alterations to it or by not publishing these alterations. A symbiotic relationship on the other hand benefits both parties. In a symbiotic relationship, the commercial organization actively takes part in the development of the project, for example by paying project developers or by publishing changes made to the project by the organization. In interviews with Swedish organizations it was found that most commercial organizations classified their relationships to open source projects as symbiotic [18], although companies might be marketing themselves in a more positive light than might actually be the case. Commercial entities would likely not admit acting in a parasitic way, for instance.

From the point of view of open source project survival, symbiotic relationships are naturally the most significant. In these relationships the commercial support can help create external incentives to project maintainers, which lowers the risk of project maintainers leaving the project due to personal reasons. In this sense, a sponsored open source project combines benefits from closed, commercial software development and the open, volunteer-based model, since voluntary development is still possible but the project development isn't entirely reliant on it.

Commercial organizations can, however, feel that active participation in open source projects isn't feasible. Some organizations avoid, for exsample, publishing the changes they have made to open source software based on the commercial interests of the organization [1]. These organizations fear that publishing modifications could lead to losing competitive advantages, like the technologies or development methods. These organizations either avoid active participation entirely or they only publish their changes when the risk of losing a competitive advantage is gone or low enough.

4.4 The effect of licensing on project popularity

All open source projects share certain principles, based on which the users of these projects are granted rights that the users of other software aren't granted. Generally speaking open source is viewed as meaning, that the software can be freely modified and both modified and unmodified version of the software can be freely distributed.

There are however differing views among open source projects about how much these rights given to users should be protected. Some open source licenses set only a few limitations on the use, alteration and distribution of the software and they work mainly as disclaimers. For example, the MIT license only requires that the copyright notice and license text be included in copies made of the software [24]. Some open source licenses impose strict requirements though. the GNU project's GPL license demands, for example, that all altered versions or software making use of GPL code must be licensed under GPL [13]. This "copyleft" clause aims to prevent anyone from removing the rights that have been granted for users of a particular project.

Restrictions do come with an effect on project popularity. More open licenses have been noted as correlating positively with popularity amongst users [23]. This can be explained, for example by the fact that less restricted code is easier to combine with other source code. For instance, commercial organizations might find the restrictions of GPL problematic.

From the point of view of project survival, a more restrictive license isn't necessarily a negative thing. Although less restictively licensed projects may enjoy wider user support, projects utilizing more restrictive licenses have been found to have higher developer participation and collaboration has been found to be more active [10]. At the same time, however, single developers have more often found to leave projects than in less restrictively licensed projects.

5 How open source projects die

There are multiple definitions for open source project success and failure. Some project can be viewed as successful if it reaches the its goals. In this thesis failure is, however, failure is defined as being separate from the goals of the project. Instead we view an open source project as a failure when it can no longer survive. We draw a parallel between the project failure and the last phase of the life cycle of an open source project, where a project withers and gets abandoned. We refer to this event as the project death.

In this chapter we cover reasons for why open source projects die and what kind of indicators of project death can be detected. We also go over the projects' ability to be resurrected.

5.1 Reasons for project death

Open source project failure can be viewed as being caused by external and internal factors [9]. A significant portion of developers of failed projects note that their project failed either due to a more successful competitor or because the project is no longer useful. A part of the developers propose failure to have been caused by outdated or difficult to maintain technologies used in the projects. In all these cases external factors have had an effect on the usefulness or maintainability of the project. In terms of Lehman's laws, these can be viewed as cases where the software has failed to adapt to changing circumstances.

Factors internal to the project contribute to the project failure as well. In some cases one of these factors can be difficult software architecture, which slows down development. More common reasons are however personal factors, such as lack of time and interest. These factors are understandably common, since much of open source development is voluntary. Open source projects are also the responsibility of single developers [2], which predisposes projects to failure due to personal reasons.

5.2 How dying projects can be detected

Waning projects can be detected by observing the activity in and around the project. The indicators of project health are related to two fundamental factors: developer interest and user interest [21]. User interest can be measured by download counts and the number of open bug reports and feature requests. Developer activity on the other hand is trivial to measure using version control systems. The connection between the two can also be studied. For example, large amounts of open bug reports and change requests might indicate a lack of developer interest even if the project has an interested user base.. User interest can be measured by download counts and the number of open bug reports and feature requests. Developer activity on the other hand is trivial to measure using version control systems. The connection between the two can also be studied. For example, large amounts of open bug reports and change requests might indicate a lack of developer interest even if the project has an interested user base.

A special feature of open source projects is, however, the fact that reaching the end of the life cycle model doesn't necessarily result in the end of the project. This is in contrast to closed source projects, where development efforts can reasonably continue as long as the people responsible for the project are still around. After active development efforts have ended, open source projects can still be resurrected. This is a natural consequence of the availability of the source code, since anybody can in theory continue developing the software.

5.3 How projects get resurrected

Withering projects can over time end up in a situation, where a large enough number of core developers leave the project that maintaining the project becomes difficult. The term "truck factor (TF)" was created in agile software development to describe this phenomenon. Truck factor refers to the smallest group of core developers, that upon leaving the project would jeopardize the continued development of the project [3]. This group of core developers is based on their contributions to the project. Developers are counted as belonging to the core developers, if their total contributions to the project account for at least 50 percent of all contributions.

The core developers leaving the project is referred to as "truck factor developer detachment (TFDD)", which can also be considered the point where an open source project gets abandoned [2]. A project in this kind of a state can be thought of as being dead or at least dying.

The TF of a project can be determined using a version control system. The TF analysis can then determine who the core developers of a project are [3]. This means that we can also detect TFDDs by tracking the latest contributions made by the core developers. If the core developers haven't made contributions within a certain time window, they can be considered lost and a TFDD has happened in the project.

The projects' ability to survive a situation like this has been studied among popular projects. The study [2] focused on projects deemed to be popular based on the number of stars given to them on GitHub. Out of these projects 16 percent had encountered a TFDD, which means that popular projects also get abandoned and failures are not limited to just unpopular projects.

TFDD isn't always the end of the project though, but it's an indication that the project has reached the end of its life cycle. Like previously stated, the life cycle isn't linear. A project can end up at the end of the life cycle, but based on developer interest it can return to a previous stage in the life cycle model.

In popular projects, the rate of survival is pretty good and 41 percent of projects with a TFDD survive [2]. In these cases the project has lost a significant portion of its core developers but new core developers have replaced those lost to continue developing the project. In most cases the project only gains one new developer, but since most projects are managed by single developers, this is usually enough to replace the one lost developer. These new developers had in slightly more than half of the cases participated in the project previously in a smaller role, but 48 percent of project resurrections involved developers entirely new to the project in terms of active participation.

Among less popular projects, a survival rate of 41 percent is naturally not to be expected. Less popular projects cannot necessarily find new motivated replacement developers. Project survival and the significant role of entirely new developers in project resurrection do indicate, that continuity in open source projects after project death is a reality.

There are also services made for abandoned open source projects, like Code Shelter [7], where volunteer developers adopt projects, that the original developers no longer have the interest or time to maintain.

6 Conclusion

Open source project life cycles form an interesting topic of study, since outsider access to source code makes possible a number of paths for the life cycle to take. This is the main reason why open source project life cycle differs significantly from that of proprietary software projects.

At the same time, open source also creates challenges to analyzing the life cycle. Online platforms, like SourceForge and GitHub have allowed easy production and publishing of open source software, which has resulted in a large quantity of projects of varying level and state. Not all of the projects on these platforms even can be considered software development and the portion of projects that can be considered practice or hobby projects is vast. This creates problems in trying to categorize and analyze data related to open source projects, and necessitates a critical approach towards the data being studied.

Some common factors have been identified among open source projects, however, that affect project life span and success. Most critical of these are developer motivation, and user interest towards the project, since interested users have a chance of becoming project developers in the future. Both of these factors can be measured and attempts can be made to affect them. Additionally other factors, like technical tools, support from commercial organizations and even the license, affect project survival.

The survival of declining projects is also very much possible, and factors related to improving the chances of this happening can be identified. Among popular projects the risk of abandonment is also quite low, and even out of the abandoned projects nearly half continue being developed by new maintainers. Less popular projects likely have lowered chances of survival, but on the flip side these projects also have fewer users and thus their abandonment has a reduced effect. However, even these projects can improve their chances of survival by lowering the barrier to entry using, for example, good documentation and a robust test suite.

Bibliography

[1] A. Alami and A. Wasowski. “Affiliated participation in open source communities”. In: *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2019, pp. 317–327. DOI: 10.1109/ESEM.2019.8870185.

[2] G. Avelino, E. Constantinou, M. T. Valente, and A. Serebrenik. “On the abandonment and survival of open source projects: an empirical investigation”. In: *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2019, pp. 328–339. DOI: 10.1109/ESEM.2019.8870181.

[3] G. Avelino, L. Passos, A. Hora, and M. T. Valente. “A novel approach for estimating truck factors”. In: *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. May 2016, pp. 1–10. DOI: 10.1109/ICPC.2016.7503718.

[4] H. Borges, A. Hora, and M. T. Valente. “Understanding the factors that impact the popularity of GitHub repositories”. In: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Oct. 2016, pp. 334–344. DOI: 10.1109/ICSME.2016.31.

[5] A. Capiluppi, J. M. González-Barahona, I. Herraiz, and G. Robles. “Adapting the “staged model for software evolution” to free/libre/open source software”. In: *Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting*. IWPSE ’07. Dubrovnik, Croatia: Association for Computing Machinery, 2007, pp. 79–82. ISBN: 9781595937223. DOI: 10.1145/1294948.1294968.

[6] W. Chen, V. Krishnan, and K. Zhu. “” Release early, release often”? An empirical analysis of release strategy in open source software co-creation.” In: *Pacific Asia Conference on Information Systems*. Ed. by J.-N. Lee, J.-Y. Mao, and J. Thong. 2013.

[7] *Code Shelter*. https://www.codeshelter.co. Luettu: 15.3. 2020.

[8] J. Coelho, M. T. Valente, L. L. Silva, and A. Hora. “Why we engage in FLOSS: answers from core developers”. In: *2018 IEEE/ACM 11th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. May 2018, pp. 114–121.

[9] J. Coelho and M. T. Valente. “Why modern open source projects fail”. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: Association for Computing Machinery, 2017, pp. 186–196. DOI: 10.1145/3106237.3106246.

[10] J. Colazo and Y. Fang. “Impact of license choice on open source software development activity”. In: *Journal of the American Society for Information Science and Technology* 60.5 (2009), pp. 997–1011. DOI: 10.1002/asi.21039.

[11] L. Dahlander and M. G. Magnusson. "Relationships between open source software companies and communities: observations from Nordic firms". In: *Research Policy* 34.4 (2005), pp. 481–493. DOI: https://doi.org/10.1016/j.respol.2005.02.003.

[12] J. Fernandez-Ramil, A. Lozano, M. Wermelinger, and A. Capiluppi. "Empirical studies of open source evolution". In: *Software Evolution*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 263–288. DOI: 10.1007/978-3-540-76440-3_11.

[13] *GNU General Public License*. https://www.gnu.org/licenses/gpl-3.0.html. Luettu: 13.4. 2020.

[14] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. "An in-depth study of the promises and perils of mining GitHub". In: *Empirical Software Engineering* 21.5 (2016), pp. 2035–2071. DOI: 10.1007/s10664-015-9393-5.

[15] A. Lee and J. C. Carver. "Are one-time contributors different? A comparison to core and periphery developers in FLOSS repositories". In: *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2017, pp. 1–10.

[16] M. M. Lehman. "Programs, life cycles, and laws of software evolution". In: *Proceedings of the IEEE* 68.9 (1980), pp. 1060–1076.

[17] Z. Liao, B. Zhao, S. Liu, H. Jin, D. He, L. Yang, Y. Zhang, and J. Wu. "A prediction model of the project life-span in open source software ecosystem". In: *Mobile Networks and Applications* 24.4 (2019), pp. 1382–1391. DOI: 10.1007/s11036-018-0993-3.

[18] B. Lundell, B. Lings, and E. Lindqvist. "Perceptions and uptake of open source in Swedish organisations". In: *Open Source Systems*. Ed. by E. Damiani, B. Fitzgerald, W. Scacchi, M. Scotto, and G. Succi. Boston, MA: Springer US, 2006, pp. 155–163.

[19] T. Mens. "Introduction and roadmap: history and challenges of software evolution". In: *Software Evolution*. Springer Berlin Heidelberg, 2008, pp. 1–11. DOI: 10.1007/978-3-540-76440-3_1.

[20] *OS market share and usage trends*. https://web.archive.org/web/20150806093859/http://www.w3cook.com/os/summary/. Luettu: 5.3.2020.

[21] J. Piggot and C. Amrit. In: *Open Source Software: Quality Verification*. Ed. by E. Petrinja, G. Succi, N. El Ioini, and A. Sillitti. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 30–44.

[22] E. S. Raymond. *The cathedral and the bazaar: musings on Linux and open source by an accidental revolutionary*. Sebastopol (CA): O'Reilly, 1999.

[23] K. J. Stewart, A. P. Ammeter, and L. M. Maruping. "Impacts of license choice and organizational sponsorship on user interest and development activity in open source software projects". In: *Information Systems Research* 17.2 (June 2006), pp. 126–144. DOI: 10.1287/isre.1060.0082.

[24] *The MIT License*. https://opensource.org/licenses/MIT. Luettu: 13.4. 2020.

[25] *Web and application servers market share report*. https://www.datanyze.com/market-share/web-and-application-servers--425. Luettu: 5.3. 2020.

[26] A. Wiggins and K. Crowston. “Reclassifying success and tragedy in FLOSS projects”. In: *Open Source Software: New Horizons*. Ed. by P. Ågerfalk, C. Boldyreff, J. M. González-Barahona, G. R. Madey, and J. Noll. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 294–307.

[27] D. E. Wynn. “Organizational structure of open source projects: a life cycle approach”. In: *Proceedings of 7th Annual Conference of the Southern Association for Information Systems*. 2004, pp. 285–290.