Automatic Means of Detecting Warning Signs in Software Evolution

ing. Martijn Endenburg
martijn.endenburg@gmail.com

July 14, 2014, 51 pages

Supervisors:	dr. Magiel Bruntink (UvA), ir. Rutger Pannekoek (VNU)
Host organisation:	VNU Vacature Media, De Persgroep Nederland
Publication status:	Published

VNU VACATURE MEDIA

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

A replicated study of

"Automatic Means of Identifying Evolutionary Events in Software Development"

original paper by Siim Karus, 2013

This thesis is submitted in fulfillment of the requirements for the degree of Master of Science in Software Engineering by ing. Martijn Endenburg, born in Hilversum, The Netherlands. As part of the part-time Master's programme at the University of Amsterdam, Faculty of Science, Science Park 904, 1098 XH, Amsterdam (www.science.uva.nl). Conducted at VNU Vacature Media, De Persgroep Nederland, Mt. Lincolnweg 40, 1033 SN, Amsterdam (www.vnuvacaturemedia.nl).

Abstract

Many organisations rely on open-source software (OSS) projects to run their businesses. Therefore, the need for the ability to assess an OSS project's survival chances is increasing. However, the events influencing OSS projects are mostly organic with little empirical validation.

This research is a replication of *Automatic Means of Identifying Evolutionary Events in Software Development* by Siim Karus and explores the use of wavelet analysis to automatically detect warning signs in OSS projects, regardless of project size or scale. Knowing warning signs helps in the decision whether to use, or to continue to use an OSS project.

Using wavelet analysis it is possible to detect events that lead to the end of code evolution of an OSS project. This study presents an analysis of the survivability of projects having such warning signs, and under what conditions wavelet analysis succeeds or fails in detecting warning signs.

Keywords: open-source software, software analytics, software evolution, project survivability, warning signs, wavelet analysis

Preface

This thesis is the result of my graduation project and concludes the part-time Master programme Software Engineering at the University of Amsterdam.

The project has been performed at VNU Vacature Media, the company at which I am employed at. VNU Vacature Media is a division of De Persgroep Nederland since May 2012. The company operates in the on-line recruitment market. It owns the largest job boards in The Netherlands and Belgium: Intermediair, Nationale Vacaturebank, Vacature.com, and References.be. The software supporting these products are mostly open-source projects, that is why the survivability of open-source software projects came to my interest in the choice of a graduation subject.

Acknowledgements

There are some people I would like to thank that contributed to my project in various ways. This thesis would not have been possible without them.

First, I would like to thank Brett Kelly, who was CTO at VNU Vacature Media at the time I started the Master's programme, and Rutger Pannekoek, who is CTO of VNU Vacature Media since 2013. Without them, I wouldn't be able to follow the Master's programme together with my job in the first place.

Second, I would like to thank my tutor at the University of Amsterdam, Magiel Bruntink, for the help and support during this project.

I would also like to thank my parents, Johan and Carla Endenburg for always supporting me in whatever I do. They always told me that "when there's a will, there's a way".

Thanks go out to my colleagues Bas, Boudewijn, Danny, Eugene, Ewout, Greg, Hugo, Jan-Willem, Jeroen, Jerry, Joeri, Johan, Marc, Marvin, Merlijn, Michiel, Nienke, Kalle, Leonard, Silvester, Tim, and Wouter (if I forget anyone, sorry for that), you were all great in supporting me in conducting my research on the days that I was physically present, but mentally absent.

Additionally, I would like to thank Siim Karus for helping me replicating his research. He has been very helpful, especially in the beginning of the project, in providing me his scripts and insights in his data.

Last, but definitely not least, big thanks to Christine Fröger, for all the love, support, patience, and encouragements to complete this thesis and for telling me to take a break every now and then.

Martijn Endenburg
Amsterdam, The Netherlands
July, 2014

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem statement

Many elements of the software development process are related to the project's progress. These elements include, but are not limited to, team composition, team size, frequency of releases, developer's activity, and developer's and user satisfaction [8, 9, 26].

A change in one or more of these elements affects the project's progress positively or negatively. If a change in one or more of the elements has a substantial effect on the project's progress, it is an *evolutionary event*; an event that changed the project's evolution.

In closed source projects, many evolutionary events are the effect of decisions made by the management [16]. The majority of closed source software projects are initiated and maintained by commercial organisations. These projects evolve according to changing business requirements, strategies, and commercial goals.

In commercial organisations, events of change in the hiring policy, or planning and deadlines, can become evolutionary to the projects affected by these changes. These events can be used to identify project scale and progress [16], which allows comparing and co-analysing different projects.

In open-source software (OSS) projects, evolutionary events are mostly theoretical with little empirical validation [16]. Changes in development activity, community, and other aspects of the OSS process are mostly organic [1]. A community of an OSS project changes continuously, often with little impact on the evolution of the project. This means that different OSS projects cannot be compared on the same scale. In other words, different OSS projects cannot be objectively assessed by their evolution data without a proper transformation of that data.

For closed source projects, the measurement of a project's success or effectiveness is critical to our understanding of the value and efficacy of management actions and investments [10]. Comparing and co-analysing OSS projects is of interest to better understand the survivability of OSS projects. The key factors that define the success or failure of an OSS project have not yet been found.

We expect to be able to find indicators of failure in the events that shape a project's evolution. Therefore, a means to identify these evolutionary events would be of use for organisations that employ OSS. For closed-source software projects in a commercial environment, these events can be tracked by the managerial history of the project. For OSS projects, this is not the case.

1.2 Motivation

At VNU Vacature Media, developers extensively use OSS projects in their day-to-day work. The projects vary from developers tools, and utilities, to full system stacks. These OSS projects include

Ansible, Apache HTTP Server, MySQL, OpenStack, OpenVPN, and Ubuntu/Linux. Besides the previously listed well-known projects, lesser known projects are also used.

Given that OSS projects are popular and contribute to the overall value of the products and services of many organisations, a means to objectively assess OSS project's survivability is of value. An assessment of the survivability of an OSS project will aid in the decision for selection and/or continuity of a project.

In the study *Automatic Means of Identifying Evolutionary Events in Software Development* by Karus [16], Karus proposed the use of wavelet analysis to automatically detect anomalies in OSS projects. Wavelet analysis is used in many fields having many purposes. One of the applications of wavelet analysis is pattern recognition in digital signal processing, such as duplication detection and face recognition in imagery [22, 28]. The wavelet algorithms process data at different scales or resolutions [13]. The big advantage of wavelet analysis is that it enables the analysis of the data at different scale levels. Another advantage is that it is based upon mathematical principles, making it fairly easy to automate.

At first glance, it may seem odd to use wavelet analysis to analyse evolution of software projects. Software evolution data comprises various metrics, such as lines of code, number of developers, number of commits, etc. The measures of these metrics are all snapshots of moments in time. These metrics can be modeled in the frequency domain of a waveform. The metrics have a functional relation to the age of a project, which fits the time domain of a waveform. This gives a natural transition of each metric to be modeled as a waveform and enables the analysis of such a waveform.

The analysis of waveforms of software evolution metrics should be able to detect anomalies in these waveforms. After investigating and comparing these anomalies it might as well be able to find the patterns that resulted from evolutionary events. A comparison with known events that had a negative effect on a project's progress may reveal that it is able to detect warning signs in software evolution.

A warning sign is defined as any pattern in the waveform that has a high chance of resulting in the *end of code evolution* of a project. The *end of code evolution* means that the source code activities of a project have stopped. There could still be commits on the project, for instance in Wiki pages, external library updates, or documentation, but no more source code changes. For a pattern being such a warning sign, it has to be similar to patterns that are confirmed to be such warning signs.

According to Karus [16], knowing warning signs will help in:

- Choosing an OSS project to implement in a business scenario.
- Making timely preparations for decommissioning an OSS project.
- Choosing a development process that best suits the aims of a software project for new OSS projects.

1.3 Research question

Q1 *Can we use wavelet analysis to find objective warning signs in open-source software projects leading to the end of code evolution?*

In order to find the answer to the main question, the following questions will be answered:

Q2 *What patterns can be found using wavelet analysis?*

Q3 *Under what conditions does wavelet analysis succeed or fail in detecting evolutionary events?*

1.4 Outline

In the next chapter, chapter 2, I describe the research approach and phases. Chapter 3 provides background information and the context of the research. Chapter 4 describes the execution of the research. In chapter 5, the results of research execution are presented. Chapter 6 discusses and analyses the results. Finally, in Chapter 7 the conclusions are presented and the research questions are answered.

Chapter 2

Research method

2.1 Research approach

The research will be conducted in the following phases:

Literature survey. As part of the research plan, relevant literature will be studied in order to get familiar with the topics touched by this study. Important topics are the ecosystem of OSS organisation and community, and the success and survival factors of OSS projects. Literature regarding wavelets, (discrete) wavelet transform and analysis, and in particular the Haar wavelet will be studied to understand its workings.

Data selection. In this phase, the OSS projects for the study will be selected. The evolution data of these OSS projects will be validated and cleansed to ensure all data for these projects is consistent.

Another criterion for the data set is that the evolution series of the projects are continuous, i.e., that the data per project does not have gaps of missing data. The discussion for this requirement is done in section 2.2.

The data set for analysis should also be representative to the total set of OSS projects in order to be able to generalise findings from the projects within the data set to the world of OSS projects in general.

Wavelet analysis. This phase is divided into two sub-steps: wavelet transform, and wavelet analysis.

First, the evolution data of the selected projects will be transformed using discrete wavelet transform (DWT). The results of these transforms are kept for further analysis. The choice for a suitable signal will be made.

Second, the results of the wavelet transform are being analysed to find patterns. A pattern is a sequence of transformed data that occurs multiple times. To be able to find patterns, a detailed definition of when a sequence is a pattern is needed. Choosing when a sequence becomes a pattern in terms of number of occurrences might influence the number of patterns found and possibly influence the types of patterns wavelet analysis is able to detect.

A distinction is made between the *detection* of a sequence and an *occurrence* of a sequence. Although the difference is slight, the distinction is needed for further analysis.

A sequence x is *detected* in a project X if it was encountered by analysing project X. Finding an *occurrence* of a sequence means that a sequence y of another project Y was found to be similar to sequence x by comparing project Y's sequences against x. A similar sequence is then recorded as a pair of sequences (x, y) detected in project X and occurring in project Y respectively.

Pattern identification. The patterns found in the Wavelet analysis step will be further analysed to find patterns that are 'warning signs'. This is a pattern that leads to the 'end of code evolution' for a project (recall Q1). The validation needed in this step is described in section 2.3.1.

Analysis and conclusions. The final phase in the study is the analysis of the results to come to conclusions and answers to the research questions (section 1.3).

2.2 Continuous series

In verification of the requirement of a continuous series for wavelet analysis, a hypothetical signal was constructed. Let the following set of pairs be the signal S to be analysed, where the first entry is the time value, and the second entry the frequency value:

$$S = \{(1, 100), (2, 150), (5, 250)\}$$

In signal S the two pairs having first entry 3 and 4 are missing. In software implementations of discrete wavelet transform, the input signal to process can be indexed. This means that whether or not the signal contains gaps, these gaps are naturally closed by indexing the signal. After zero-based indexing, the signal S will look as follows:

$$S' = \{(0, 100), (1, 150), (2, 250)\}$$

The wavelet transformation can perfectly be done using S' as input signal. The output series of coefficients are indistinguishable from other signals without gaps. However, the patterns found will be invalid as it ignored intermediate data during similarity analysis.

A further investigation can be done on how to fix such gaps. The use of four different methods of 'guessing' the data were explored and the results are the following:

last observation carried

$$S_1 = \{(1, 100), (2, 150), (3, 150), (4, 150), (5, 250)\}$$

mean

$$S_2 = \{(1, 100), (2, 150), (3, 200), (4, 200), (5, 250)\}$$

piecewise constant interpolation

$$S_3 = \{(1, 100), (2, 150), (3, 150), (4, 250), (5, 250)\}$$

linear interpolation

$$S_4 = \{(1, 100), (2, 150), (3, 183), (4, 216), (5, 250)\}$$

The method *last observation carried* results in a signal S_1 with the same values as second entry for points 2, 3, 4 and 5, which shows a flat line when plotted in a graph.

Filling the gaps with *mean* values, will also result in a signal S_2 with same values as second entry. It is similar to last observation carried but with the difference that the mean method is exclusive, whereas 'last observation carried' is inclusive. Using mean, the number of second entries having same values will therefore be 2 observations lesser than using the last observation.

The *piecewise constant* interpolation results in a signal S_3 that, when plotted, has two shorter flat lines. The repeating values are 'snapped' to the closest neighbour.

Finally, the *linear interpolation* will result in a signal S_4 that linearly flows towards the next known value. The advantage of this technique is that there will not be a flat line in the resulting signal. However, the disadvantage is that resulting values possibly are not even close to what was present in the complete signal.

The flat lines introduced using the above techniques may be detected as no change in frequency during the period that the data was missing. This is undesired in this research as it can be mistaken for a pattern that stopped code evolution.

Moreover, the signal resulted from linear interpolation could be mistaken for growth or decay in frequency, depending on the direction.

The success of one of the above methods of closing a gap depends on the size of the gap. The smaller the gap, the lesser impact it has on the reliability of the detection of a pattern for that particular signal. The larger the gap, the more the impact on reliability. Either way, guessing data will add fictional data and, in general, having gaps in the evolution data and closing the gaps by adding data will negatively influence the reliability of detecting patterns.

Therefore, only projects of which a continuous series of data is available will be selected for this study.

2.3 Validation

2.3.1 Dead projects

In order to validate the patterns found and in particular to know if a pattern represents an evolutionary event leading to the end of code evolution, dead projects have to be identified.

A project is considered 'dead' if it has one or both of the following properties:

1. the project has had 0 contributors in code for the last 12 months;
2. the project has had 0 'lines of code' (LOC) churn[1] for the last 12 months.

The above properties specify that the project has had no code activity for the past year, meaning the code evolution of the project has stopped. If the LOC churn of a series of data points equals 0 (zero), this implies that the difference in 'lines of code' (LOC) over that same series also equals 0 (i.e., there was no growth or decay in code size).

According to this definition, changes in documentation, wiki pages, external library updates, and other non-code changes are still allowed for dead projects.

To compare dead and alive projects from the data set, the above definition can be used to identify the projects from the data set that are dead. The projects that are not 'dead' by this definition are considered to be 'alive'.

To be sure that the projects from our data set that comply to the definition of dead actually have no activity beyond the time boudaries of the data set, manual validation of these findings have to be done. In verifying if a project is dead, I will manually consult the project's website, its source code repository, and possibly other sources.

2.3.2 Warning signs

To recognise patterns that are warning signs, a selection of patterns occurring in dead projects will be made for further analysis. The best candidates for patterns being a warning sign are patterns occurring near the end of the evolution of a dead project.

Knowing these patterns, selecting the projects having these patterns will be done to find out if these projects have a higher chance of dying than projects that do not show such pattern. An estimation of the survivability of projects having that kind of pattern will be done to evaluate this.

[1] LOC churn: the sum of LOC added and LOC deleted [11].

Chapter 3

Background

3.1 The OSS organisation

Open-source and closed-source software projects have much in common on the software level. However, on the organisational level, the two worlds differ a lot.

The members of Free/Libre OSS (FLOSS) projects participate voluntarily as the projects are open to everyone who wishes to contribute. In closed-source/proprietary projects, contributors are selected by the management through a hiring policy and salaries [1].

Decisions in OSS projects are mostly made by the core developers or project leaders without a formal structure. Decisions can be made by a voting system. In closed-source projects, the decisions are made by the organisational management in a strict/rigid structure.

The releases in OSS projects are often done frequently, even minor improvements, usually not on fixed dates. As opposed to closed-source software, with infrequent major releases on fixed dates.

The differences as stated previously emphasise the fact that there is no uniform and clear structure in OSS projects that enables the comparison of multiple projects regarding activity and progress.

This organisation structure is also referred to as *bazaar style*, because it is open for everyone to contribute [1, 17, 24]. The bazaar style is the opposite of the *cathedral style*, where there is a group of elite members who decide which contributions make it in the next release.

3.2 Software evolution

3.2.1 Feedback loops

The term *software evolution* refers to the ever-going change a software project goes through. This law of continuing change is the first of the eight laws of software evolution by Lehman [21].

Lehman's laws are intrinsic to what he calls *E-type* systems [20]. E-type software systems "*... are programs or applications that solve a problem or implement a computer application in the real world.*"[21]. This is the type of software that is taken under analysis for this study.

Software does not evolve by intrinsic feedback loops, like evolution in plants and animals, but by extrinsic feedback. Evolution by intrinsic feedback is related to feedback pressures from within the organism. Software is not a living organism and thus changes in the software are related to feedback pressures from the outside world [21].

There is a need for continuing adaptation for E-type systems. The reason for this is twofold: on the one hand there is the ever-changing requirements and needs for the users of the software. People who use the software find bugs, require more features, and so on. On the other hand there is the ever-changing environment or world the software 'lives' in. If depending software is changing, the change propagates to the system at hand.

Both types of extrinsic feedback result in changes of the original system and 'feeds' the evolution of the software.

3.2.2 Maintaining complexity

The extrinsic feedback is not the only source of change to a software system. Continuously adapting to environmental and users' needs eventually pays its toll. As the second law of Lehman says that "*... as a software project is evolving, the complexity of the program increases unless work is done to maintain or reduce it*" [21]. This means that, due to changing the software as a result of changing needs in the environment or users, the software needs to change even more in order to keep it below a certain level of complexity.

3.2.3 Activity

The fourth law of software evolution by Lehman [21] – the law of organisational stability – denotes that the "*... average effective global activity rate on an evolving system is invariant over the product life time.*".

The most interesting finding of Lehman in this law is that the invariance holds regardless of managerial decisions. This means that the activity of a healthy software system is constant over the life time of the product.

Lehman's laws of software evolution are valid for both closed source software and open-source software. These laws form the foundation of what we understand as software evolution.

The evolution of a software system comprises various measures on different moments in time. For instance, the lines of code metric over a project's life time is a way to measure the evolution of lines of code in a software project. More on the measures of software evolution in the next sections.

3.3 Success factors

In the survey of OSS projects by Androutsellis-Theotokis *et al.* [1], project success is defined from three perspectives:

Prerequisites for success

The prerequisites for a good chance of a successful project are divided in four categories: the first is the need for *starting points* for a project. The starting points are threefold; (1) a certain interest is required in the mind of one person or a group of developers; (2) the audience the software is targeting should be clear from the beginning; and (3) the natural language of the project may play an important role.

Second is the *community organisation*. A meritocratic culture would be the best chance for a successful project. That is, a culture that is ruled by the people who deserve it. The leaders come from the inside of the community and are by definition supported by that community.

Third is the aspects that contribute to the prerequisites for success are the *technical* environment and infrastructure as well as programming language. This contributes to the efficiency of the software development process. The choice of external project dependencies affects the success of a project under development.

And fourth, a prerequisite for project success is the *open-source perspective*. The licensing scheme and the definition of an effective and reasonable business model are issues that should be clear from the beginning of a project.

Sustainability

Ensuring the sustainability of a project that has reached a certain level of success. At the system level this means achieving identified goals, update documentation and testing material. The importance of software maintainability plays an important role in the sustainability of

a successful project. This shows similarities to the second law of Lehman's laws of software evolution [21].

On the community level, sustainability is evenly important. This comprises the level of developer and user satisfaction. This can be achieved through continuously evolving and taking advantage of the developers' skills and competencies [1].

Indicators for success

The indicators to distinguish a successful project from other, less successful ones are of interest by many researchers. A lot of indicators for success or success factors have been identified, but the ultimate key indicators for success have not yet been found. The next section will elaborate more on this subject.

3.3.1 Indicators of success

Success is not a binary state but requires a measurement; a certain extent of prosperity.

The survey by Androutsellis-Theotokis *et al.* [1] identified success indicators in four perspectives. A project that has progressed into advanced development stages is more likely to be a healthy project. Such projects have an active community of developers.

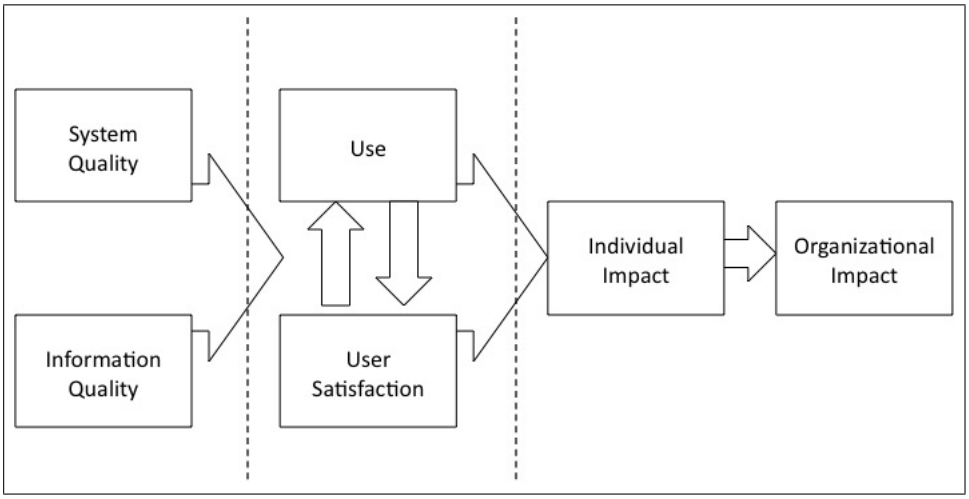
The number of downloads is an indicator for success; a frequently downloaded project is likely to be successful.

Frequent and visible development or bug fixing activities are indicators of a healthy and successful project.

Another indicator for project success is the robustness of a project. Indicators for a robust project are also indicators of success, according to Androutsellis-Theotokis *et al.* [1].

Delone and McLean [9] defined a model of information systems success (Figure I). In this model, success factors are identified on multiple levels: system and information quality, use and user satisfaction, and individual and organisational impact. In the revised article [10] the authors evaluate the model using more recent empirical results on the application of their model across different studies.

Figure I: The Delone & McLean model of information system success (1992)



©1992 Delone and McLean [9]

It was found in between four and sixteen studies that there are significant relations between "system use", "system quality", and "information quality" on the one hand, and "individual impacts" on the other hand.

System use was typically voluntary in sixteen studies and measured as frequency of use, time of use, number of accesses, usage pattern, and dependency.

System quality was measured in five studies in terms of ease-of-use, functionality, reliability, flexibility, data quality, portability, integration, and performance.

Information quality was measured in four studies as accuracy, timeliness, completeness, relevance, and consistency.

Individual impact was measured in terms of job performance, decision-making performance, job effectiveness, and quality of work and work environment.

A study by Crowston *et al.* [7] took the idea of Delone and McLean [9, 10] and examined the vision of system development underlying the model and project them onto OSS projects. The authors identified additional measures that might be used for OSS project success. The vision is a process model that has just three components: the creation of a system, the use of the system, and the consequences of the system use.

Crowston *et al.* [7] suggest that given the large number of abandoned projects, the completion of a project is an indicator for success. However, it is hard to define what it means for a project to be completed. Therefore, the progress of a project from one development stage to the other is considered an indicator for success. This is similar to the findings of Androutsellis-Theotokis *et al.* [1].

Another indicator for success is whether the project achieved its goals. The problem is that OSS projects often do not express their goals and do not have formal requirement specifications. One goal that is often implicit in OSS projects is developer satisfaction. This, in turn, is similar to findings of Androutsellis-Theotokis *et al.* [1].

Measures regarding the process of software development are identified. Crowston *et al.* [7] states in accordance to Lehman [21] that software development is an ever-going activity. A number of possible indicators of success are suggested by Crowston *et al.*:

Number of developers
The number of developers appears important for the continuity of a project and could therefore be an indicator of success.

Level of activity
Seems to be more important than the number of developers. The developers' contribution to the project in submitting code, and bug reports.

Cycle time
The time between releases, or cycle time, might be an indicator for success. The norm of the OSS community in general is 'release early and often', which implies an active release cycle is a sign of a healthy project.

Employment opportunities
The motivation of OSS developers might be an indicator for success. Developers may participate in an OSS project to improve their employment opportunities. It can be measured as their salary, or jobs acquired.

Individual reputation
Related but not quite the same as the employment opportunities it is considered an indicator of success if developers are willing to participate because they will be rewarded with reputation in the community. Measurement, however, is very difficult. Earlier studies did not succeed in relating the earning of reputation to the success of a particular project.

Knowledge creation
An effect that can be measured by surveying the developers for their perceived learning on a project. It seems important for developers that they learn during their participation in a project.

Although important, the previous list of measures is by no means an exhaustive list of measures for OSS project success. The disclaimer by Crowston *et al.* [7] states that by no means they claim that any of these measures actually measures project success.

In another study, three years later, conducted by Crowston *et al.* [8], the authors extended the earlier study using Free/Libre Open-Source Software (FLOSS) projects. The authors demonstrated the application of the measures as listed in the previous paragraphs in 122 projects from SourceForge[1]. The paper shows the challenges for mining source code repositories and OSS communities.

One measure that was revisited is the *number of developers*. Crowston *et al.* suggests the 'number of developers' as a measure is a flawed number, because it aggregates the numbers of developers joining and the leaving a project. A developers 'churn' (i.e., sum of developers joined and left) or a 'tenure' (i.e., developers time on the project) would be more appropriate in measuring team size or composition.

3.4 Evolution in OSS

3.4.1 Project transition

A research by Nakakoji *et al.* [24] studied the 'natural product evolution' of OSS projects using four typical OSS projects. Three categories of software were identified:

Exploration-oriented
Sharing innovations and knowledge.

Utility-oriented
Satisfying an individual need.

Service-oriented
Providing a stable service.

Nakakoji *et al.* found that the community of an OSS project co-evolves with the system. Additionally the authors found that OSS projects evolve from one category to the other. Typically, an exploration-oriented project evolves into a service-oriented project. A successful exploration-oriented project attracts many developers which speeds the code evolution, mostly not in favour of the exploration-oriented project goals. This increases the risk of the project to be forked and the original project to be abandoned.

Utility-oriented projects also typically mutate into service-oriented projects. According to Nakakoji *et al.* the service-oriented variant of a utility is a merge of combined forces of multiple utility-oriented projects providing similar features.

3.4.2 Project characteristics

A study by Koch [19] analysed the evolution of more than 8,621 OSS projects from SourceForge. The projects are analysed by the following metrics:

Lines of code (LOC)
At the initial commit of a project, the LOC is computed for all source files. For any subsequent commits, the LOC added, and LOC deleted is recorded. The difference between LOC added and deleted can be used to compute the cumulative LOC at any given time.

Revision
The submission of a single file by a single programmer.

Maturity
An indicator of maturity is given by SourceForge ranging in seven possible values: planning,

[1] `http://www.sourceforge.net`

pre-alpha, alpha, beta, production/stable, mature, and inactive. This metric is equal to the development stage indicator of Androutsellis-Theotokis *et al.* [1] and Crowston *et al.* [8].

The projects are analysed by the previously mentioned metrics. Findings of the analysis are mostly done on growth characteristics. A result was that a higher number of active programmers has no negative influence on productivity in mean output per programmer in LOC. The same counts for output per programmer in revisions.

Furthermore, it was found that projects with super-linear growth are larger in LOC, are more active in number of revisions, and have more programmers, than projects with linear growth. However, the previous findings do not always count for super-linear versus sub-linear growth.

A study conducted by Beecher *et al.* [2] on the characteristics of a total of 300 FLOSS projects from 6 distinct repositories used metrics to measure and characterise the projects, similar to Koch [19].

One additional metric was used by Beecher *et al.* in comparison to the other studies:

Duration
: Measured as the number of days over which the project was developed. This was evaluated using the earliest and latest available dates in the source code repository for the project. It gives the actual age of the project, at least since the moment the community decided to use a source code management system. It is a good indicator of the time-span of the project.

It was found by Beecher *et al.* that repositories differ from each other in terms of product and process characteristics. The authors compared two groups of projects characterised by repository. The first group having projects from Debian, KDE, and GNOME, and the second from RubyForge, Savannah, and SourceForge. Both groups show consistently different characteristics in size, duration, commits, and committers. These metrics were used to identify success relative to other projects in the same group and across groups. A growth in size, a relatively longer duration, relatively more commits and committers would be 'more successful'.

Beecher *et al.* found that the projects in the first group achieve better results compared to the second group. Projects that made the transition from the second to the first did better on previous metrics than before the transition. This implies that the type of repository influences the project success. Projects in repositories Debian, KDE, and GNOME are in general more successful than projects in repositories RubyForge, Savannah, and SourceForge. This is probably due to the stricter rules for a project to enter one of the repositories in the first group and the free nature of the repositories in the second group.

3.4.3 Software evolution prediction

A study by Goulão *et al.* [12] used seasonal time analysis to predict the evolution of OSS projects. The authors use an ARIMA[2] model that fits time series data (i.e., a sequence of data points measured at successive, uniformly spaced data points) of change requests.

The findings of Goulão *et al.* were that not all projects show seasonal patterns. The ARIMA model was trained on one project's history data (in this case, Eclipse) to be able to accurately predict the evolution of that project. After training, the model could actually adequately predict evolution in the long-term.

Although, the prediction of evolution using the ARIMA model is not subject to this study, but it does show that the predictability of the evolution of OSS projects exist.

3.5 Project survivability

Several studies have been conducted regarding the survivability of OSS projects [25, 26, 29].

[2] auto-regressive integrated moving average

Raja and Tretter [25] defined and evaluated a measure of survivability of OSS projects. The measure the authors found is *viability*: *the ability of a project to grow and maintain its structure in the presence of perturbations*. The viability is defined along three dimensions:

Vigor
: The ability of a project to grow/evolve over a period of time. Measured as number of versions released per year.

Resilience
: The ability of a project to recover from disturbances. Measured as response time to artifact requests.

Organisation
: The structure exhibited in the project. Measured as average mutual information. That is, the structure of the artifact management process reflects the organisation of the project.

The viability of a project was tested in 232 random projects from SourceForge, of which 136 projects were used as analysis sample, and 96 as validation sample.

The three dimensions are mathematically formulated to compute a coefficient representing the 'value' of either of the dimensions. A higher value indicates a stronger position in the dimension.

The authors found that viability is consistent with reality. In addition, the three dimensions – vigor, resilience, and organisation – are different attributes contributing to viability.

In a study by Samoladas *et al.* [26] on the survivability of OSS projects, the authors performed an analysis on the duration of 1,147 OSS projects from various sources using a similar, but slightly different, interpretation of the duration metric of Beecher *et al.* [2]:

Duration
: The age of the project in months since the birth month (i.e., the first month the project was present in the data). Projects are split into two groups: the active, and inactive projects. Inactive projects have had less than two commits for two months in a row.

Samoladas *et al.* employed a Kaplan-Meier estimation of the survival function and the hazard function of the projects. Analysis was performed by using the duration metric on all projects, and on projects category as the projects were categorised in 25 distinct categories.

The findings by Samoladas *et al.* were that there is an effect of project category (i.e., application domain), and number of committers on the duration of a project. For instance, adding developers to a project increases the survivability of a project. The authors proposed two main research issues to be addressed in the future. The first is to add more projects to the study with possibly a different categorisation as it may influence the results. And second, the effects of more project parameters, such as programming language should be examined. This is not trivial since typical more than one language is used in a project.

Wang [29] investigated survival factors of FLOSS projects obtained from SourceForge. Wang conducted the investigation at two stages of a project's life cycle. These stages are:

Initial stage
: The initial period of time, typically the first year of existence when there is no public release of source code.

Growth stage
: The period of time after the first public release of the core source code, typically after the first year of existence.

Wang defined the following independent variables and measures for OSS project survivability:

Survival in initial stage
: The likelihood of a project to survive the first year. Measured by the ability to remain active on hosting website.

Survival in growth stage
The likelihood of a project to survive after the first year. Measured by the ability to produce the first release of source code, adoption rate, and technical improvements.

User/developer participation
The extent to which users/developers are involved in the development of the software. Measured by involvement of users and developers in bug submissions, bug fixes, etc.

Service quality
The degree to which the project team satisfies the needs and requirements of the users and developers. Measured by the implementations of feature requests, bug fixes, and the degree to which users' needs are satisfied.

License restrictiveness
The degree of restrictions on the users and developers to redistribute software derived or modified from the project. Measured by the classification of the OSS license type of the project.

Targeted users
Whether a project is targeted at technical users. Measured by the targeted user group; whether the user group is software developers or systems administrators.

Internal network size
The size of the network of developers from within the project, measured by the number of developers.

External network size
The number of members and members of other OSS projects that developers of the project are involved with.

Quality of external network
Measured by the SourceForge ranking of the affiliated projects.

The findings of Wang [29] were that the level of OSS projects' user/developer participation will positively impact the project's survival at initial stage. However, it is only marginally supported that the same holds for the project's survival at growth stage.

The level of service quality will positively impact the project's survival at both the initial and growth stages. The license restrictiveness has no significant impact on the survivability of a project. This is in contrast to the observations regarding open-source perspective of Androutsellis-Theotokis *et al.* [1].

Furthermore, network sizes do not impact the survivability, but the quality of the external network does at both stages.

Wang [29] also found that warning indicators can be found in six factors of OSS project success: developer participation effort, developer service quality, license restrictiveness, targeted users, external network size, and external network quality. A decay in one or more of these factors might be a warning sign.

3.5.1 Project causes of death

According to Karus [17] there can be two reasons for an OSS project to die:

Loss of popularity
A product loses its share because of other similar projects. Or a project has lost its value because of technical advances on other platforms, or deprecated hardware/software dependencies.

Reaching maturity
A project might reach maturity in case it will be used without changes for the foreseeable future.

The second of the two reasons is in contrast with Lehman [21]'s laws of software evolution as noted in section 3.2.

Besides the previous two reasons, one more reason can be thought of using the study by Nakakoji *et al.* [24]. The *forking* of a project, i.e., starting a new project from an existing one. The two projects become substitutes and eventually one will replace the other. Both projects evolve separately.

The effect of these events will ultimately result in *abandonment* by the developers, the users, and the entire community.

Detecting such events and 'predicting' the end of code evolution of OSS projects is the subject of this study. One proposed method to accomplish this is using wavelet analysis on evolution signals of OSS projects.

3.6 Wavelet analysis

Karus explored a method known as *wavelet analysis* to analyse software evolution data [16, 17]. The wavelet analysis interprets evolution data as a series of signals and is able to find sequences in this signal. The sequences of multiple projects can be compared in order to find recurring patterns.

A study by Hindle *et al.* [14] explored the use of Fourier transform and analysis to find repeating patterns or processes hidden in the logs of change events of OSS projects. However, Fourier transform seems not the right tool to use for analysis of evolution time series data.

Wavelet analysis is similar to Fourier analysis with the difference that wavelet transform deals with time and frequency information, and Fourier transform deals with frequency information only. Wavelet analysis has the advantage over Fourier analysis that it can deal with non-smooth ('choppy') signals, as opposed to the limitation of only sines and cosines [13].

3.6.1 Waveforms and wavelets

Wavelets are functions that satisfy certain mathematical requirements and are used in representing data or other functions [13].

A waveform, or wave, is a signal (time-series) in a two-dimensional space. Waveforms are used to model many types of signals, such as audio, electromagnetic (light), gravitational, and quantum mechanical waves.

In many models of waveforms the two dimensions typically represent the time and frequency domains. An audio signal is an example of a waveform modeled that way.

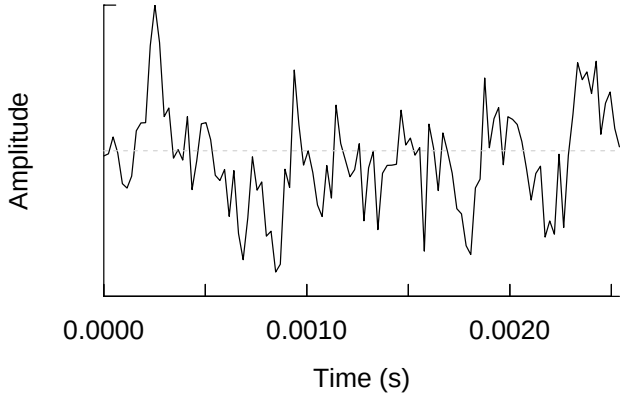
In visualisations of waveforms, the time domain is often plotted on the horizontal axis, and the frequency domain on the vertical axis. In the example of an audio wave (see Figure II), the frequency domain may represent the amplitude, or the frequency, and the time the duration of the amplitude or frequency value.

However, the time domain is not specifically bound to time intervals only. A less intuitive approach could be to model frequency in the time domain. That way the frequency relation to amplitude of a signal can be analysed, but the time information is lost.

3.6.2 Discrete wavelet transform

To be able to analyse and compare wavelets of different lengths and scales, a way to transform the signal is needed. Discrete wavelet transformation is the operation of applying a filter function, or set of filter functions (also known as 'filter bank') to the wavelet. 'Discrete' because the signal is segmented into samples; which number is possibly finite, possibly infinite. The discrete wavelet transformation is a way of sampling the signal at different intervals giving a natural means of scaling the signal [16].

Figure II: Example of an audio wave



In digital signal processing, the category of methods that discrete wavelet transform belongs to, there is a phenomenon called the *uncertainty principle*. It means that the shorter the signal, the harder it is to identify the signal. On the contrary, the longer the signal, the higher the accuracy in identifying the signal. However, it is impossible to identify a signal with 100% accuracy as there is always a degree of uncertainty in identifying a signal. This is a fundamental principle in digital signal processing and it highly influences the ways the method can be used.

High-frequency signals vary quickly and are shorter in nature than low-frequency signals that vary slowly. Therefore, in order to identify high-frequency signals, time resolution is needed, and to identify low-frequency signals, frequency resolution is needed. It is not possible to zoom in on both parts of the signal at the same time. Zooming into time will compromise on frequency and *vice versa*.

Wavelet transform is the transformation of signals (time-series) by decomposing the signal into wavelet/shift coefficients, and scaling/filter coefficients based on filter functions (i.e., filters) [16].

Shifting/translating is the operation of moving the wavelet in the time domain and filtering/dilating is the operation of scaling the wavelet in the frequency domain. These operations are illustrated in Figure III and Figure IV respectively.

Figure III: Shifting/translating a wavelet

(left) The input signal: 1 second sine wave
(right) The signal delayed by 1 second

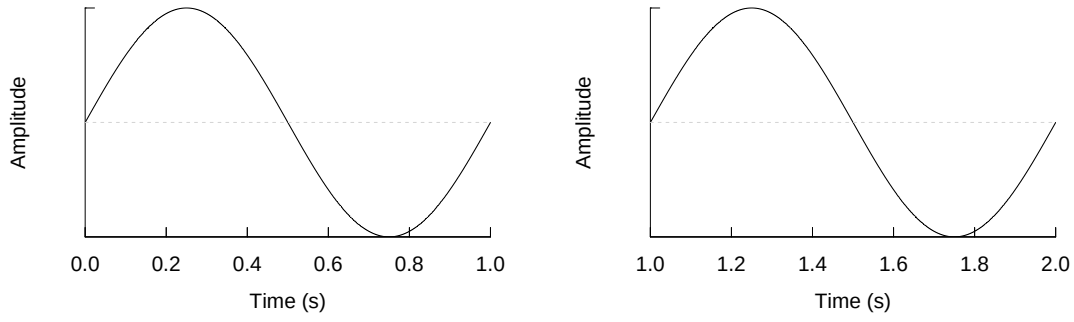
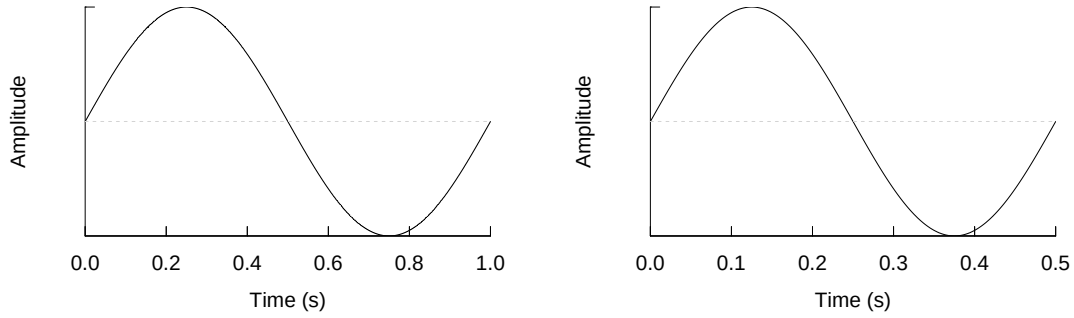


Figure IV: Filtering/dilating a wavelet

(left) The input signal: 1 second sine wave
(right) The signal scaled by 0.5 to a $\frac{1}{2}$ second sine wave



Wavelet transform has proven to be important in signal processing thanks to its inherent properties which allow comparisons at different scales and shifts [16]. Compared to other time series analysis techniques, the main advantages of wavelet transformations in the analysis of signals are:

- Wavelet/shift coefficients allow fuzzy matching as differences in details are 'smoothed out'.
- Filter/scale coefficients allow detection of small anomalies in series.
- Decomposition levels make series of different lengths or scale comparable.

3.6.3 Wavelet functions

A wavelet function is a function that defines a wavelet [28]. Many wavelet functions exist and differ largely in complexity and applicability depending on the signal to be processed.

A wavelet can be defined in the following ways, given that T is the set of time values of the signal, and F the set of frequency values of the signal [13].

Wavelet function (mother wavelet)
$\Psi : T \to F$
$\Psi(t) = f$, such that $t \in T$ and $f \in F$.
A bijective function mapping T onto F and producing the shape of the wavelet.

Scaling function (father wavelet)
$\Phi : F \to T$
$\Phi(f) = t$, such that $t \in T$ and $f \in F$.
A bijective function mapping F onto T and producing the scale of the wavelet.

Scaling filter
A low-pass filter of length $2N$ and sum 1. A high-pass filter can be calculated as the quadrature mirror filter of the low-pass filter.

3.6.4 Haar wavelet

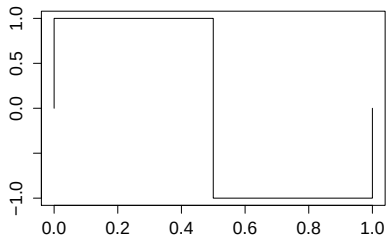
The Haar wavelet is a member of the Daubechies family of wavelets, based on the work of the Belgian mathematician Ingrid Daubechies [13]. The Daubechies wavelets is a family of orthogonal wavelets defining a discrete wavelet transform. All wavelets of the Daubechies family can be entirely defined by their scaling filter.

The Haar wavelet is a Daubechies wavelet and has a filter length of 2 and is therefore also referred to as the Daubechies-2 (D2) filter. It is the simplest wavelet of the Daubechies family and is defined by the mother wavelet function as shown in Figure V [27].

Figure V: The Haar functions

(a) The Haar wavelet function (mother wavelet)

$$\Psi(t) = \begin{cases} 1 & 0 \leq t < 1/2, \\ -1 & 1/2 \leq t < 1, \\ 0 & \text{otherwise.} \end{cases}$$



(b) The Haar scaling function (father wavelet)

$$\Phi(t) = \begin{cases} 1 & 0 \leq t < 1, \\ 0 & \text{otherwise.} \end{cases}$$

In 1910, the Hungarian mathematician Alfred Haar introduced Haar functions. The Haar transform is one of the earliest examples of what is known now as a compact[3], dyadic[4], orthonormal[5] wavelet transform [27]. The Haar function is the simplest and oldest orthonormal wavelet with compact support.

3.6.5 Discrete Haar transform

The Haar filter function varies in scale by splitting the input signal using different scale sizes.

Having a signal over the domain from 0 to 1, the Haar transform divides the signal into two wavelets that range from 0 to 1/2 and from 1/2 to 1. Then the division can be repeated giving four wavelets that range from 0 to 1/4, from 1/4 to 2/4, from 2/4 to 3/4, and from 3/4 to 1 [13].

The original signal is decomposed successfully into components of lower resolution. The decomposition can be repeated as long as the resolution of the original signal allows. In the case of the Haar filter this means as long as the number of resulting coefficients is larger than 2 (i.e., the filter length).

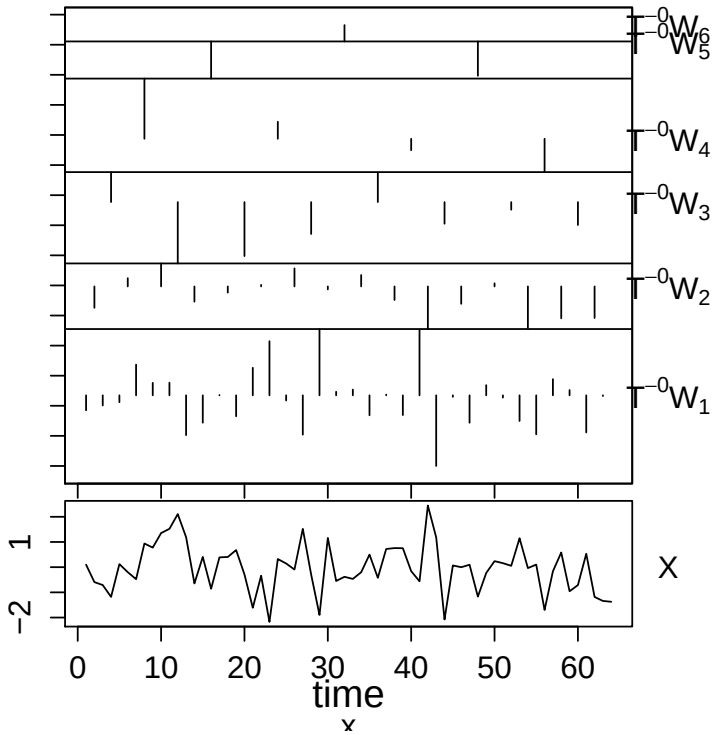
In each scaling/filtering step (i.e., decomposition) the Haar function adds more detail to the wavelets in the current level. The Haar filter captures the differences between scale levels. These resulting coefficients can be used to compare signals regardless of scale or length [13].

Wavelet transform using the Haar filter is illustrated in Figure VI. The figure shows a stacked plot of a signal (lower most graph) and the transformed signals at each level of decomposition (the stack).

[3]Compact support: the ability of a function to be scaled smoothly.
[4]A dyadic function scales in powers of 2.
[5]Orthonormal: domain and range are orthogonal and of equal length.

Figure VI: Discrete wavelet transform using Haar filter

$W1, \ldots, W6$ represent the levels of decomposition in reversed order, the lower most graph shows the original signal. The coefficients plotted at $W6$ (top most graph) are generated in the first step of decomposing. Each level from $W6$ to $W1$ adds detail from the original signal. The highest level ($W1$) is closest to the original signal. No more detail is left from the original signal that can be added in any further step.



The spikes shown in Figure VI represent the coefficients found at the relevant level. The number of coefficients increases by a factor of 2 at each level. In the figure, the original signal consists of 64 samples, therefore, the signal can be decomposed to a maximum of $64 \log_2 = 6$ levels.

The coefficients should only be compared to coefficients of the same type. The scale/filter coefficients are incomparable to the wavelet/shift coefficients, because that would mean comparing time with frequency which leads to invalid relations between signals.

Wavelet transform using the Haar filter is widely used in other fields. For example, as a way of digitalising an analogue signal in A-D converters, pattern recognition in image and video processing, face recognition, image processing, data coding, multiplexing, digital filtering, digital speech processing, voice controlled computing devices, robotics, and compression mechanisms [18, 27, 28].

Chapter 4

Research

4.1 Data selection

A total number of 250 OSS projects will be used for this study. This number is somewhat arbitrary, but was chosen to have a larger data set than the initial study conducted by Karus [16], as a larger data set yields more trustworthy results. However, the research still needs to be feasible within the given time constraints.

Several steps have been taken prior to the selection of this set of 250 projects. The steps are described in the following sections.

4.1.1 Data gathering

The data for this research is gathered from Ohloh[1]: *"Ohloh is a free, public directory of Free and Open Source Software and the contributors who create and maintain it."* [15]. At the time of this writing, Ohloh tracks more than 660,000 OSS projects varying in all ranges of size, and popularity. The most popular projects currently being Apache HTTP Server, Apache OpenOffice, Apache Subversion, Bash, Firebug, Linux Kernel, Mozilla Firefox, MySQL, PHP, and Ubuntu.

For this research, the data provided by a tool by Bruntink [4] is used. This tool, "*OhlohAnalytics*", was developed as part of the replication research by Bruntink [3], and provides a validated and cleansed data set of 12,360 OSS projects, collected from Ohloh in July 2013.

Initial validation and cleansing

The OhlohAnalytics tool performed initial analysis, validation, and cleansing of the data by detecting inconsistent values and removing these records from the data set. Additionally, extra data fields are derived or aggregated from and added to the raw data for convenience.

The complete list of data fields provided by OhlohAnalytics is shown in Table I. The column 'Source' specifies if the field is either 'Raw' data (i.e., directly available from Ohloh), or if it is added by the OhlohAnalytics tool as derived from an operation on one or more other fields. In the latter case, the fields are listed.

The result of the data gathering and cleansing step is a consistent data set of evolution data of 10,811 projects.

4.1.2 Data validation

Continuous series

Although the data set should be consistent after validation and cleansing, another validation step is needed prior to the selection of the 250 projects for the study.

[1] `http://www.ohloh.net`

Table I: Monthly data fields

Field	Source	Description
project_name_fact	Raw	The name of the project at Ohloh.
Activity		
abs_loc_growth	loc_added_fact, $loc\_deleted\_fact$	The number of 'lines of code' (LOC) that the project has grown (or shrank) current month.
blanks_added_fact	Raw	The number of blank lines added to source text current month.
blanks_deleted_fact	Raw	The number of blank lines deleted from source text current month.
comments_added_fact	Raw	The number of lines of comments added in source text current month.
comments_deleted_fact	Raw	The number of lines deleted from source text which are comments; for current month.
commits_fact	Raw	The total number of commits made current month.
contributors_fact	Raw	The total number of contributors who made at least one commit current month.
ind_loc_growth	loc_fact, abs_loc_growth	The relative growth of the project measured in LOC for current month.
loc_added_fact	Raw	The number of LOC added current month.
loc_deleted_fact	Raw	The number of LOC deleted current month.
Analysis		
age_in_months	$month\_fact$, $year\_fact$	The age of the project in months measured since the first data point; starts at 0.
age_in_years	age_in_months	The age of the project in years measured since the first data point; starts at 0.
cumulative_commits_fact	$commits\_fact$	The total number of commits since the first data point (i.e., where $age\_in\_months = 0$).
main_language_fact	Raw	The programming language having the highest LOC value for the project in current month (XML and HTML are ignored).
month_fact	Raw	The month value of current month's analysis. Extracted from $udpated\_at$ field.
year_fact	Raw	The year value of current month's analysis. Extracted from $updated\_at$ field.
Size		
blanks_fact	Raw	The total number of blank lines in source text in current month.
comment_ratio_fact	Raw	The fraction of net lines in source text which are comments; for current month.
comments_fact	Raw	The total number of lines in source text which are comments; for current month.
loc_fact	Raw	The total number of LOC current month.

The evolution data of the 10,811 projects contained gaps. A continuous series of data is required to be able to analyse time series for a project (see section 2.2). Therefore, a number representing the *fraction of continuity* of the evolution data per project was calculated. For each project the difference between the minimum and maximum values of the *age_in_months* fact is taken, added by one, giving the expected number of data points for a project. The computation of the fraction of total evolution data is done for each project.

After the fractions of the total evolution data for each project were calculated, the projects were filtered and only those projects that have all data points between minimum and maximum *age_in_months* are kept. This reduces the total number of projects to 6,418.

Minimal sequence length

A time series of 1 (monthly) data point is not analysable over time and incomparable to larger projects. The threshold of at least 12 monthly data points is chosen to minimise noise in the evolution data that may be caused by too young or unstable projects.

After this selection the data set contains 5,986 projects.

Sample selection

For the selection of a sample of 250 projects, the tool created by Nagappan *et al.* [23] at Microsoft Research is used. This tool takes a - possibly atomic - sample set and selects additional projects to add to the sample that increase the overall representativeness of that sample. The tool scores projects by two metrics: total lines of code, and yearly contributors count.

The tool iteratively selects 250 projects and adds it to the sample. Each additional project is selected by its score to maximally increase the representativeness of the sample as a whole, compared to the master data.

The master data is a list of 20,028 projects tracked by Ohloh delivered with this tool. A pre-filtering of the master data was done to make the tool select only the projects that appear in the data set of 5,986 projects.

4.1.3 Dead projects

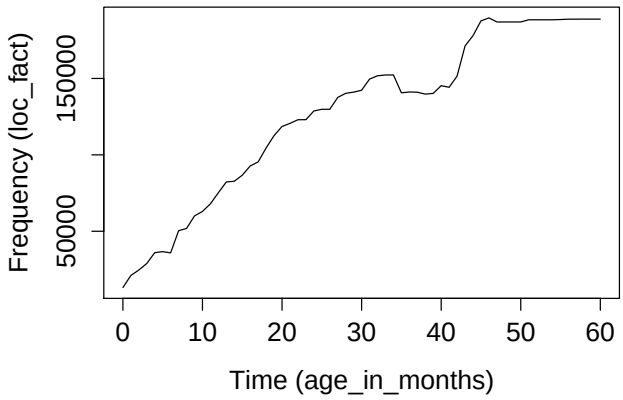
In the search to find patterns that are warning signs leading to the end of code evolution, the projects that satisfy the definition of a dead project (see also section 2.3.1) were extracted from the 250 projects.

4.2 Wavelet transform and analysis

4.2.1 Project signals

The evolution of the 250 projects is modeled as signals for wavelet transform and analysis. The constructed project signals are modeled with *age_in_months* in the time domain, and *LOC* in the frequency domain. Figure VII illustrates a project's signal.

Figure VII: LOC signal of project #19012



The choice for 'lines of code' as time series is done because of its simplicity and intuitive interpretation. Additionally, LOC over time is a way to measure the growth of the project.

The *LOC* metric used in this study for the transform and analysis is slightly different from the loc_fact that is available at Ohloh. The goal for the signal processing is to be able to detect patterns in a project's code activity. This means that all changes that occur in source files are of interest. Therefore, the *LOC* metric is composed in the same way as in the original study by Karus [16] and is constructed as the sum of loc_fact, $comments\_fact$, and $blanks\_fact$.

For the wavelet transform and analysis, I have used the R Statistics Suite with packages "wavelets", "chron", and "zoo". The Wavelets package contains implementations of discrete wavelet transform functions, including the Haar filter; Chron provides functions for working with chronological data series; and Zoo eases indexing of and working with indexed data.
The Haar filter is used by Karus [16] because of its simplicity and ease of interpretation. As this study is a replication of Karus' study, the choice of using the Haar filter was made.

The R scripts used for the steps of wavelet transform, sequence identification, and grouping, are based on the scripts created and used by Karus in the initial research. Small adjustments have been made to make the scripts compatible with the data set.

4.2.2 Wavelet transform

The first step in the analysis of time series of software evolution is the wavelet transform. During this step discrete wavelet transform using the Haar filter is applied on a project's signal. The results comprise the coefficients at each level of decomposition which are saved for further analysis.
The coefficients are not the same as data points. The data points represent monthly facts from a project, and the coefficients represent values found at a certain level of decomposition from the wavelet transform. More specifically: the coefficients are values resulted from shifting or scaling the signal. For more details on wavelet transform and analysis see section 3.6.

4.2.3 Wavelet analysis

Similar sequence identification

The coefficients resulting from the wavelet transform are analysed to find *similar sequences*. A sequence is defined as a series of coefficients at a particular level of decomposition, with a sequence length between 3 and 65 (inclusive) coefficients.

The sequence length thresholds were chosen by Karus [16] in the original study to distinguish a sequence from an ordinary set of coefficients if the sequence is very short (length 1 or 2). Or, in case the sequence is very long, the ability to distinguish the sequence from the complete signal, hence the maximum length of 65.

In the search for similar sequences, the sequences of one project at a time are analysed against other sequences. A sequence is considered 'similar' if at least one other sequence was found of which the values of the coefficients are equal with respect to an allowed deviation of 0.005. This number is a configuration made by Karus in the original study. The choice was made to keep it the same in order to get similar results. A lower or higher value will yield fewer or more similar sequences respectively.

The process of finding similar sequences constructs ordered pairs of sequences from the one project to another (see section 2.1). The first entry of the pair is the sequence and project of which the sequence was first discovered (i.e., detected), the second is the sequence and project of the similar sequence (i.e., occurrence). Similar sequences may be found within one project, but can also be found across projects.

The properties of a similar sequence are described in Table II.

Table II: Similar sequence properties

Property	**Description**
Coefficient type	The type of coefficient; either W (wavelet/shift), or V (filter/scale).
Project0	The identity of the project the sequence was detected in.
Project1	The identity of the project having the similar sequence.
Level0	The compression level at which the sequence was detected.
Level1	The compression level at which the similar sequence was found.
Start0	The index of the coefficient the detected sequence starts.
Start1	The index of the coefficient at which the similar sequence starts.
Length	The length representing the number of coefficients of the sequence.
Maximum deviation	A number representing the precision of similarity as the maximum deviation between coefficients of project0 and those of project1.
Sequence	The list of coefficients representing the sequence of project0.

Similar sequence grouping

Not all of the sequences found in the previous step are of equal value. Many of the sequences considered similar are contingent or represent trivialities. In the sequence grouping step, the similar sequences are taken as input to find 'patterns'.

A sequence is considered a 'pattern' if it appears at least 3 times in the sequence data. Each pattern is assigned an identification number to be used in further analysis.

This process constructs key-value pairs of sequences having the detected sequence as key (i.e., the pattern), and a set of similar sequences (i.e., occurrences) as value. Additionally, meta data is added to the grouped sequences. The properties of a pattern, in addition to the sequence properties from Table II, are shown in Table III.

Table III: Pattern properties

The list of sequence properties also applies to patterns.

Property	**Description**
ID	The identity of the pattern.
Project count	The number of projects containing at least one similar sequence.
Occurrences	The number of occurrences of the pattern.
Project list	The list of projects having a similar sequence.
Sequences	The list of similar sequences making the pattern; defined as a list of 3-tuples of project ID, level, and start index.
Has dead	A boolean value indicating whether the project list contains at least one dead project.
Has alive	A boolean value indicating whether the project list contains at least one alive project.

4.3 Pattern classification

Given the dead projects in the data set, all patterns detected in the dead projects are selected. Patterns detected in dead projects can be hints of evolutionary events that lead to the death of the project.

The patterns found in dead projects will not occur all at the same moment in the lifetime of the projects. Therefore, three types of patterns for dead projects are introduced.

Type A pattern A pattern is a Type A pattern if and only if:

- it is detected in a dead project;
- for all occurrences in dead projects, the end of the sequence matches the end of the dead project evolution.

For example: if the evolution of a dead project P starts at point x and ends at point y, then a pattern detected at point x' and ending at point y in project P is a type A pattern.

Type B pattern A pattern is a Type B pattern if and only if:

- it is detected in a dead project;
- for all occurrences in dead projects, the end of the sequence does not match the end of the dead project evolution.

Characterising the patterns according to the above definitions will leave a group of patterns that *weakly* comply to both definitions.

Type AB pattern A pattern is a Type AB pattern if and only if:

- it is detected in a dead project;
- for at least one occurrence in dead projects, the end of the sequence matches the end of the dead project evolution;
- for at least one occurrence in dead projects, the end of the sequence does not match the end of the dead project evolution.

The above types partition the total set of patterns detected in dead projects into three disjunctive sets. Although the classification of the patterns into these types mention only the occurrences in dead projects, the patterns may occur in alive projects too. However, occurrences in alive projects are not taken into account for the classification.

4.4 Survivability

In the search for patterns that are warning signs the patterns of type A will be the most interesting, because these patterns occur near the end of evolution of dead projects. These patterns could indicate an evolutionary event that has lead to the end of code evolution for the dead projects it appears in.

To determine whether the chances for a project to die increase when having a type A pattern, two groups are made:

G0 The group consisting of projects not having an occurrence of a type A pattern.

G1 The group consisting of projects having an occurrence of a type A pattern.

Both groups contain dead and alive projects, are of equal size, and are disjunct by definition.

The selection of projects for G1 is done by selecting all projects having an occurrence of a type A pattern. The selection of G0 (i.e., the control group) is done by using the tool from Nagappan *et al.* [23] to make the control group representative to the whole set of projects.

For each dead project it is recorded at what age the project died. The dying age in months for dead projects and the maximum age in months of alive projects are taken as time indicators for the projects. The state whether a project is 'dead' is taken as event status for the Kaplan-Meier estimation of survival function for the projects.

The results of survival estimation are presented in section 5.5.3.

Chapter 5

Results

5.1 Data

The data set of 250 projects was gathered in July 2013. It contains monthly data points for each project up to June 2013.

The data set was tested for representativeness using the tool by Nagappan *et al.* [23]. The data set scored a 99.5% representativeness to the tool's master data of 20,028 projects tracked by Ohloh.

The final data set contains monthly evolution data of 250 distinct projects having a total of 22,943 data points. The oldest project having 321 monthly data points, the youngest having 14 monthly data points. The first data point in the set is of October 1986. The last data point is of June 2013. The total OSS development time in the data set spans 22,943 months, which is 1,911.9 years.

5.2 Dead projects

A total of 38 (15.2%) projects complied to the definition of a dead project as defined in section 2.3.1. Manual verification of the 38 potential dead projects using the project's websites, source code repositories, and commit history up to April 2014, revealed that 21 (8.4%) are still complying to the definition of a dead project by April 2014.

The 21 dead projects and their *age_in_months* at the moment of death as a result of the verification are shown in Table IV.

Table IV: Dead projects' final age (in months)

Project	Final age	Project	Final age	Project	Final age
317799	2	586805	14	4614	75
587198	3	360279	30	41745	80
588411	5	322065	37	155830	84
589515	7	11389	46	325178	92
587204	8	12053	68	4007	120
585077	9	3085	71	15700	121
587571	11	307140	71	12547	142

All, except one, of the projects in Table IV are dead because it was abandoned by the community of contributors. Except for project #587204, which is still receiving updates, but at very slow and sporadic intervals. However, since the updates do not involve code activity, it is considered dead.

When looking into project #317799, which is the youngest in this set and died in its second month, it shows that this project has had a history of the slightest change in its lines of code evolution. The project has had a total number of 7 commits during the time it is being tracked by Ohloh (since September 2011). A total of 4 contributors have worked on the project since it was tracked. The most recent commit was done in October 2011.

Project #12547 is the oldest. It died after 142 tracked months and has had its most recent commits in February 2012. A total of 10 contributors have worked on this project since May 2000.

The special case project #587204 is the only project that is not abandoned by its (entire) community. The project has had 3 contributors over its lifetime. One of them is still active every now and then. The most recent commits were done in January 2014, the commits before that were in June 2012. The commits involved updates in documentation, and the creation of a configuration file for a continuous integration server. These commits do not involve code changes.
The project died in July 2012, after 8 months since the first data point tracked.

From the initial 38 projects, the remaining 17 projects complying to the definition of dead (section 2.3.1) appeared to be still alive. These 17 projects are evaluated:

- For 9 of the projects activity is very low or rapidly decreasing. Their community is slowly but surely abandoning the project as can be seen by a decrease of 35% to 55% of contributors, and/or commits.
- 5 of the projects are migrated to another source code repository. For these projects the tracking information is not updated at Ohloh and are lost. The tracking is stopped from the moment the project is migrated. The tracking and analysis can be recovered by updating the source code locations at Ohloh, but for this study the project is out of sight.
- The 3 projects that have had no activity in a year (i.e, were dead), but after that show little increase of activity are more difficult to explain. Manual evaluation showed that there exist similar projects outside the data set that eventually die, but there are also similar projects that eventually got 'resurrected'.

5.3 Sequences and patterns

The wavelet transform of the LOC signals of 250 projects resulted in 22,943 data points decomposed up to 7 levels into wavelet/shift and filter/scale coefficients.

Table V: Similar sequences count

Total	1,669,448	100.000%
In filter/scale coefficients	1,669,432	99.999%
In wavelet/shift coefficients	16	0.001%

As shown in Table V, the similar sequence identification found a total of 1,669,448 sequences that occurred at least 2 times. A mere 16 sequences were found in wavelet/shift coefficients, the other 1,669,432 sequences were found in filter/scale coefficients.

The second wavelet analysis step aggregated similar sequences across multiple pairs together to form patterns. The detection of these 'popular sequences' found 16,049 patterns. The patterns consist only of sequences of scale/filter coefficients.

The patterns occurred between 5 and 1,512 times across the projects. On average, a pattern occurs 104 times. A single pattern occurs in at least 1 and at most 204 projects (36 projects on average). The pattern length is between 4 and 19 coefficients (on average 6) across decomposition levels 3 to 7.

On the 3^{th} level, the average pattern length is equal to 4 coefficients. At every subsequent level the average pattern length is doubled. This is expected as there are twice as many coefficients available. This means the same pattern is detected at multiple levels.

Table VI presents the numbers of patterns detected and occurring across projects. Recall the distinction between detecting and occurring patterns from section 4.2.3.

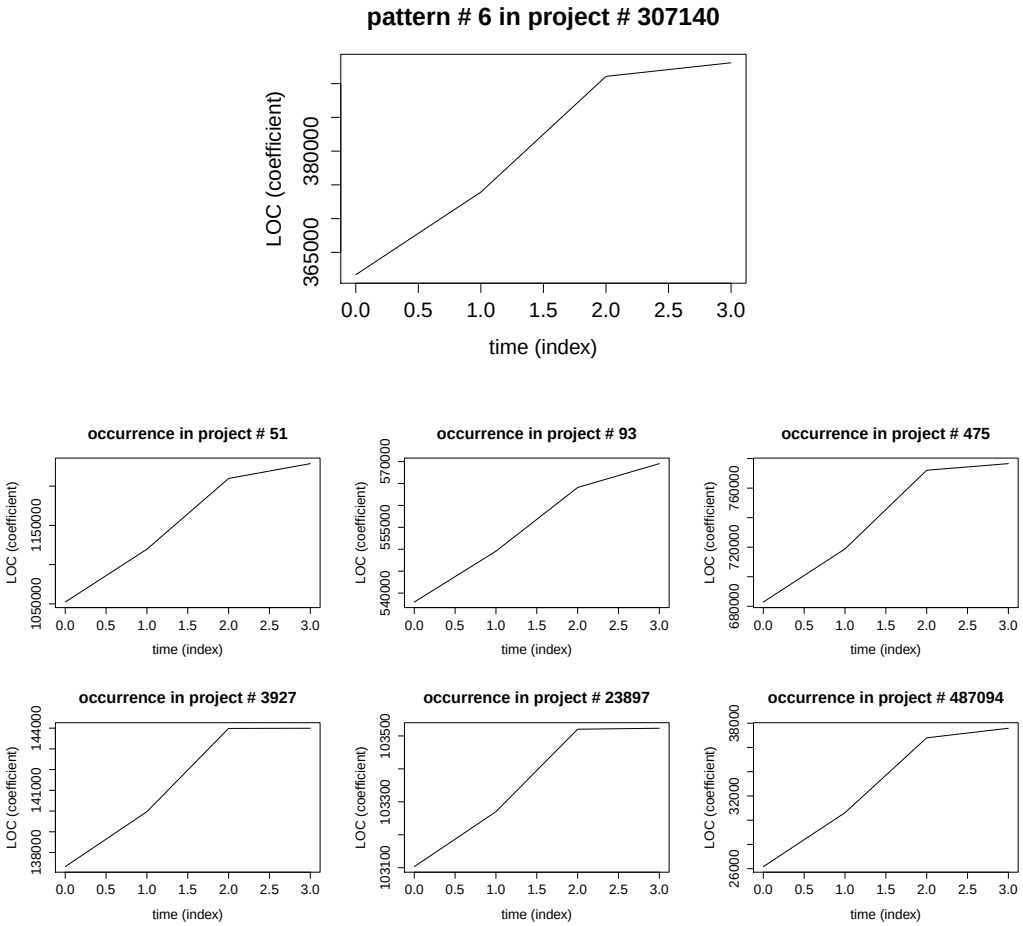
Table VI: Patterns detected and occurring

Patterns detected			**Count**	
total			16,049	100.00%
in dead projects			1,084	6.75%
in alive projects			14,965	93.25%
Patterns detected in dead projects	**Occurrences**		**Count**	
total	111,848	100.00%	1,084	100.00%
in dead projects only	111	0.10%	16	1.48%
in both dead and alive projects	111,737	99.90%	1,068	98.52%
Patterns detected in alive projects	**Occurrences**		**Count**	
total	1,560,883	100.00%	14,965	100.00%
in alive projects only	37,333	2.39%	3,390	22.65%
in both dead and alive projects	1,523,550	97.61%	11,575	77.35%

As can be seen from Table VI, 16,049 patterns were detected across 250 projects, which is 64.2 patterns per project on average; 14,965 patterns were detected across 229 alive projects (65.3 patterns on average per project); and 1,084 patterns detected in 21 dead projects (an average of 51.6 patterns per project). The averages do not differ significantly over the projects.

An arbitrary pattern that was found is shown in Figure VIII. The graph shows the similarity of the pattern with its occurrences across multiple projects. Time is indexed and LOC is represented as coefficients. This is due to the fact that wavelet transform scales and filters these measures and thus no longer represent real-world values such as months or actual lines of code.

Figure VIII: Example of a pattern found during wavelet analysis



5.4 Pattern classification

The classification of the patterns was done to distinguish different types of patterns to be able to find evolutionary events. For this, the patterns detected in dead projects were taken and classified according to the definitions in section 4.3. It is expected that the dead projects have the highest chance of keeping a pattern indicating an evolutionary event leading to the end of code evolution.

A total of 1,084 patterns detected in dead projects is classified. The sizes of the pattern type subsets are shown in Table VII. For each subset, it is stated in how many projects a pattern of that type was found.

Table VII: Pattern types

Type	Pattern count		Occurrences	Dead projects		Alive projects		Total projects
A	13	1.20%	167	8	8.6%	85	91.4%	93
B	382	35.24%	10,741	18	8.8%	186	91.2%	204
AB	689	63.56%	100,967	21	9.3%	205	90.7%	226
	1,084	100.00%						

5.5 Survivability

As described in section 4.4, the survivability of the projects having a type A pattern is estimated.

The groups G0 – projects without an occurrence of a type A pattern –, and G1 – the projects having an occurrence of a type A pattern – both contain 93 projects. Projects in G1 are selected by the characteristic of the occurrence of a type A pattern. This comprises 93 different projects as can be seen in Table VII. The projects for the control group G0 are selected as not occurring in G1 and being representative to the whole set of 250 projects.

In the next sections both groups of projects are evaluated.

5.5.1 Group G0

Group G0 contains 4 dead projects. These projects are shown in Table VIII. As stated in section 5.2, all dead projects except #587204 are abandoned by their communities.

Table VIII: Dead projects in group G0

Project	Final age	Patterns found			
586805	14	AB	6	B	1
587198	3	AB	9	B	0
587204	8	AB	15	B	4
588411	5	AB	12	B	0
Total			**42**		**5**

Table VIII shows that the four dead projects of group G0 have no pattern of type A, which is true by the definition of the group. However, they do all have occurrences of patterns of type AB. This means that there are patterns detected in these four projects that occur near and last until the end of code evolution of these projects.

The other 89 projects of G0 stay alive. Their ages vary between 13 and 255 months.

5.5.2 Group G1

The dead projects in group G1 are shown in Table IX. The column 'TTL' ('time to live') states the number of months the project has lived between the 'diagnosis' of the type A pattern and the death of the project. The 'Diagnosed' column states the age in months the pattern was found.

Table IX: Dead projects in group G1

Project	Final age	TTL	Diagnosed	Project	Final age	TTL	Diagnosed
3085	71	20	51	155830	84	41	43
4007	120	36	84	307140	71	70	1
4614	75	22	53	317799	2	1	1
11389	46	37	9	322065	37	30	7
12053	68	13	55	325178	92	4	88
15700	121	120	1	360279	30	1	29
41745	80	22	58	587571	11	8	3

All projects in Table IX are abandoned (see section 5.2). One quarter of the projects in the table live 9 months after a type A pattern was found. Another quarter lives 36 months after diagnosis. The dead projects in group G1 on average live up to 30 months since a type A pattern was found.

Project #15700 is an outlier; an occurrence of a type A pattern was found when the project was 1 month old, but it lived for another 10 years before it died.

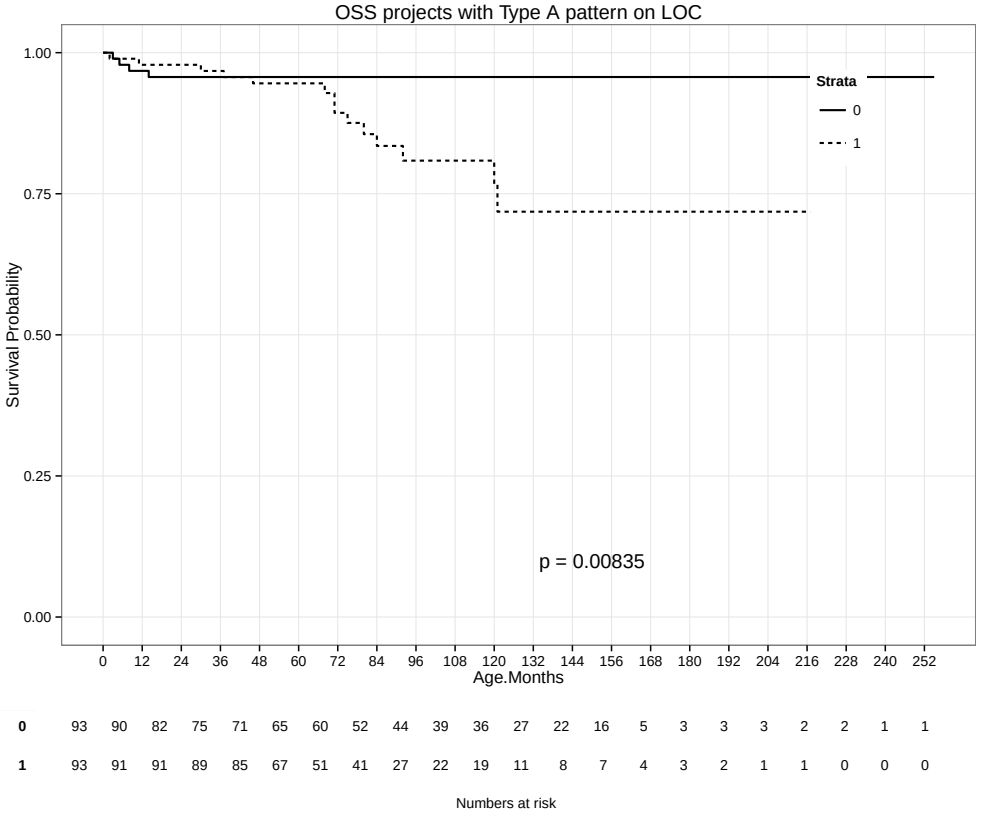
The majority (79) of the projects in group G1 are still alive. The 75 projects that have been 'diagnosed' of a type A pattern have had the occurrence in their first month of age. In only 4 projects the pattern was found later: 23, 33, 52 and 62 months.

5.5.3 Estimation of survival

The Kaplan-Meier estimation of survival function for the projects is shown in Figure IX, having the survival probability on the vertical axis, and the age in months of projects on the horizontal axis.

The numbers at the bottom, divided in two rows representing group 0 and group 1, represent the number of projects alive per group at the corresponding point on the horizontal axis.

Figure IX: Kaplan-Meier estimation of the survival function of projects regarding type A patterns



The Kaplan-Meier estimation of the survival function of the projects as shown in Figure IX visualises the survival probability of the projects with and without the occurrence of a type A pattern.

The figure shows a p-value of $p = 0.00835$. This is the log-rank value representing the equality of the two groups. In general, a p-value of $p > 0.05$ indicates the two groups are incomparable (i.e., the two groups do not show significant differences). Thus, a p-value of $p = 0.00835$ indicates comparable groups. Typically, the larger the groups, the lower the p-value. It seems a group size of 93 projects is large enough to estimate survivability of projects having an occurrence of a type A pattern.

Figure IX shows that the projects of G0 have a survival chance of approximately 93% after the first year. In the first 12 months, the survival chances drop to around 86%. After that, the survival chance remains stable for projects in G0.

More than half of the projects in G0 lives longer than 84 months (7 years) against 72 months (6 years) in G1. A quarter of the projects in G0 lives longer than 144 months (12 years), against 108 months (9 years) in G1. Two projects of G0 outlives all projects in G1 after 18 years.

During the first year, the projects of G1 have 3% higher survival chance than those of G0: approximately 96%. It is after 4 years that the projects of G1 have a lower survival chance than those in G0. Between approximately 3.2 and 3.8 years of age, projects in both groups show equal survival chances.

Chapter 6

Analysis and Discussion

6.1 Patterns

6.1.1 Detection

There are more than 100,000 times more similar sequences found in the scale/filter coefficients than in the wavelet/shift coefficients. The large difference in number of sequences found between the two types of coefficients is due to the fact that the LOC metric is a cumulative metric. The typical trend of a LOC signal is growth. This makes finding similar sequences using shift coefficients (i.e., along the time axis) less likely.

No patterns were detected in shift coefficients. This can be explained by the fact that the 16 similar sequences in shift coefficients are not similar within the same group of sequences.

Additionally, the shift coefficients are incomparable to the filter coefficients because they were found in a fundamentally different way of signal transformation. Mixing both types of coefficients would neglect the way the coefficients were found and invalidate the patterns comprising sequences of both types of coefficients.

6.1.2 Similarity

The patterns that were detected show strong similarity among signals. The similarity was demonstrated in Figure VIII in section 5.3. The figure presents an arbitrary pattern and its occurrences across different projects. The figure also demonstrates how wavelet transforms 'smooths out' differences in details by scaling and filtering the signals.

One of the properties of the Haar function is that it is reversible. This means that it is possible to pick any wavelet on any level of decomposition that resulted from Haar transform and use it to re-compose the original signal. This is a useful property for testing a pattern. Figure X shows the type A pattern #3256 projected onto the original signals of two projects.

Figure X: Type A pattern #3256 in two projects

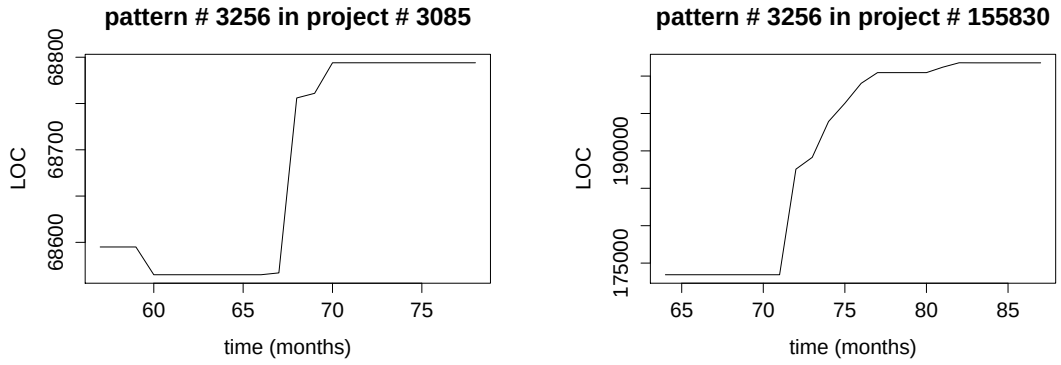


Figure X shows the wavelets of the occurrence of type A pattern #3256 in two dead projects. The pattern starts near, and lasts until the end of code evolution of the projects. This figure shows that the pattern ends in a stagnation of LOC change. The shapes of both wavelets are similar, even though the wavelets are not scaled in this figure – project #3085 is about 30% smaller than project #155830 when comparing the LOC values.

Manual evaluation using the commit logs of these two projects revealed that:

- The rise in LOC in the signal of project #3085 started in its 67^{th} month of age. It is caused by an update to a file with an addition of about 200 lines of code. This does not seem much, but relative to the project it is significant and sufficient to shape the pattern.
- The rise in LOC in the signal of project #155830 starting in its 72^{nd} month appears to be caused by a fix of a broken build and happened after 25 months of no LOC changes. As far as can be told from the commit logs, many lines of code (approximately 10,000) had to be added in order to fix the build.

The flat line starting at month 70 in project #3085 and at month 83 in project #155380 indicate a stagnation in LOC changes and lasts for 15 and 12 months respectively.

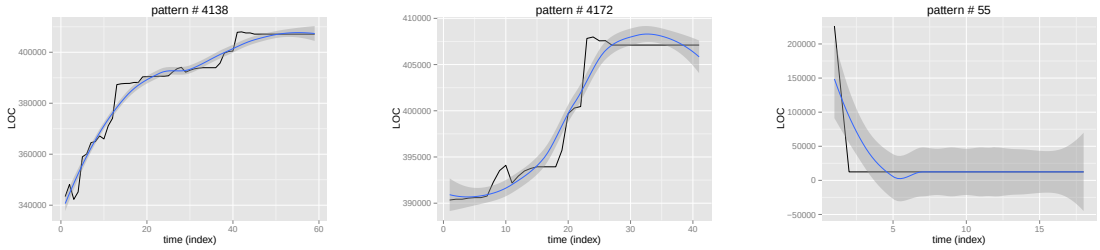
6.1.3 Warning signs

The classification of the patterns detected in dead projects (see section 4.3) was needed to distinguish possible warning signs from other recurring events. The type A patterns were expected to be the best candidates for representing warning signs because of the property that they last until the end of code evolution of a dead project.

Type B patterns are the patterns not lasting until the end of code evolution of dead projects. The typical shape of the type B patterns found in LOC signals in this study are sub-linear growth in LOC. The other two common shapes of type B patterns is an almost super-linear growth sine shape and super-linear decay. The latter shows a free fall in LOC. Figure XI illustrates all three shapes.

Figure XI: Typical type B patterns

(left) Sub-linear growth
(center) Super-linear/sine-shaped growth
(right) Super-linear decay



Occurrences of type AB patterns are found anywhere in the evolution of dead projects. This means that they may or may not be a candidate for representing a warning sign. The difficulty is the fact that the AB patterns do not occur in a consistent manner: in one project it may be an indicator for end of code evolution, while in another project it does not.

False-negatives

When looking back at the projects selected for survival analysis in group G0 (section 5.5.1), 4 out of 93 projects *not* having a type A pattern died. The patterns indicating the impending end of code evolution were not detected as such. However, all 4 projects had occurrences of type AB patterns.

Evaluation showed that the 42 AB patterns from the four projects in group G0 all show a stagnation in LOC change. The difficulty is that the type AB patterns also occur somewhere else in a dead project's evolution, which cannot be identified uniformly as possible warning signs.

False-positives

Taking the projects in group G1 (section 5.5.2), 79 out of 93 projects were still alive by April 2014. The alive projects already lived longer since diagnosis than the dead projects in the same group. This means that not all patterns of type A can be interpreted as a warning sign.

It can be a possibility to treat AB patterns also as possible warning signs, because of the property that in *some* projects the pattern occurs near, and lasts until the end of code evolution. This, however, will yield more false-positive results because a pattern occurring anywhere in a project's evolution could hardly be an indicator of the end of that evolution. On the other hand, treating the AB patterns explicitly *not* as possible warning signs – as in this study – will yield more false-negative results.

6.2 Survival analysis

The Kaplan-Meier estimation of the survival function of the projects as shown in Figure IX suggests that projects in group G1 – the projects having an occurrence of a type A pattern – die earlier than the projects in group G0 – the projects without an occurrence of a type A pattern.

The four dead projects of G0 die in their first 14 months (see section 5.5.1). This causes both strata in the curve to diverge a little and gives the impression that the projects of group G1 have a higher chance of survival than those of group G0, at least for the first 3 years.

Two of the dead projects in G0 are small ($< 5{,}000$ LOC), the other two quite large ($> 150{,}000$ LOC). However, the larger two have had their LOC value since the very first data point. This suggests that these projects were developed outside of source control and at one time everything was committed at once. The total number of commits in the history of the projects confirms this: both projects have less than 30 commits in their entire 2 year history.

The smaller two projects have had 40 and 60 commits in respectively 3 and 4 years time. The four projects could therefore as well be outliers that cause a slight distortion in the curve.

6.3 Threats to validity

The following aspects were found which may lead to threats to the validity of the results.

Construct validity

Missing historical data – In the analysis of project's evolution data, only the data provided by the OhlohAnalytics tool [4, 3] was used. The data before the first data point in the set is not taken into account. It is possible that certain evolutionary events happened before the first point. These events were not detected as they lay beyond reach of this study.

Missing most recent data – The data provided for the study has data points until June 2013. Therefore, not the most recent data of the projects is used.

LOC as activity indicator – The use of LOC (defined as lines of code + comments + blanks) as a measure of project activity could be false. The LOC will not change between two months whenever the amount of code deleted is equal to the amount added. In that case, the churn would be twice the LOC added/deleted, but the LOC will stay the same.

When patterns are detected that show a stagnation in LOC, the suggestion would be that the project's activity is decreasing. However, it might also be the case that by coincidence it seems activity is decreasing, but in reality lots of activity has been going on.

Data source – Using only one data source (Ohloh) may influence the value of the metrics. Ohloh did the analysis of the project's source code repositories. Several studies have shown that the data provided by Ohloh needs a thorough examination and cleansing before it can be used [3, 4, 5]. The data was initially validated and cleansed to make it consistent, but no actual verification on the correctness of the data was conducted.

Internal validity

Selection bias – The selection criteria for the data was to have at least 12 data points. It turned out, no project in the set was younger than 14 months. It might be that these younger projects show significance in the ability to detect evolutionary events, possibly refute or confirm findings.

External validity

Replication – It seems hard to precisely replicate the study as there are a multitude of possible configurations to be made that may, in the end, influence the results drastically. Such as, the definition of similarity between sequences (what deviation is allowed, what is the difference of coincidence and a reoccurring sequence), and the definition of a pattern (how many occurrences, minimum, and maximum length).

Furthermore, there is the interpretation of what a 'warning sign' should look like on a pattern level.

6.4 Future work

A next step in the research on the use of wavelet analysis for detecting warning signs in software evolution would be to use a larger data set. It would be interesting to know if the findings of the type A patterns will be consistent. The whole usable data set for this study contains 5,986 projects (section 4.1).

The analysis could be done on other signals. In this study, only LOC was used, but many other signals can be constructed. LOC evolution measures code activity to a certain extent, but LOC churn could reflect code activity better. To use LOC churn properly, the LOC modified fact is also needed.

Other useful metrics for finding warning signs could be team size, developer's churn (the number of developers added and removed from the team), contributor's code activity, bug reports, and defects per kLOC.

Chapter 7

Conclusions

What patterns can be found using wavelet analysis?(Q2)

The patterns detected in this study using wavelet analysis on LOC signals comprise similar sequences of LOC series within and across projects. The waveforms of these patterns vary in all kinds of shapes. Type A patterns all show a stagnation in LOC changes, that is, between very few and no changes relative to the project size. Whereas types B, and AB patterns vary between super-linear and sub-linear growth or decay (see Figure XI).

Under what conditions does wavelet analysis succeed or fail in detecting evolutionary events?(Q3)

The events underlying the patterns vary widely and the event cannot directly be derived from a pattern. A pattern is merely the symptom or result of an event. Determining the event that caused the pattern requires an in-depth analysis of the project and the time period comprising the pattern. Moreover, various patterns may show similarities, but similar patterns often do not refer to the same event.

In general, the ability to detect a pattern as being a group of similar sequences within or across projects, depends on the similarity of the sequence between other sequences within the same analysis. Therefore, it is important to have a data set that is large enough and representative to the world of OSS projects to be able to detect patterns that can be generalised as being warning signs.

Another aspect that contributes to the success or failure of wavelet analysis for software evolution is that the input signal should be free of gaps, as discussed in section 2.2. In the case an input LOC signal for analysis is not continuous between the start and end boundaries of the signal, the wavelet analysis will detect patterns that are not trustworthy, and wavelet analysis may find more false-positive evolutionary events.

The classification of the patterns in this study was done to reduce the number of patterns that may indicate warning signs. The chosen characteristics for this classification influenced the reliability of finding warning signs.

Setting up stronger characteristics for the classification treats fewer patterns as possible warning signs. This yields more false-negatives as some warning signs might be missed.

On the contrary, setting up weaker characteristics for the classification treats more patterns as possible warning signs which yields more false-positive results.

Can we use wavelet analysis to find objective warning signs in open-source software projects leading to the end of code evolution?(Q1)

In this study, I have shown that wavelet analysis is able to find patterns that increase the chances of a project to end. There exists a relation between a pattern of type A and the death of a project.

However, this relation was not found to be a causal relation.

As pointed out in the previous paragraphs, wavelet analysis detects patterns that are symptoms or results of an event in the software development process. Using wavelet analysis as a technique to find warning signs can be done, but it requires classification and verification in order to identify patterns as such.

Bibliography

[1] Stephanos Androutsellis-Theotokis, Diomidis Spinellis, Maria Kechagia, and Georgios Gousios. Open Source Software: A Survey from 10,000 Feet. *Technology, Informations and Operations Management*, 4(3–4):187–347, 2010. doi:10.1561/0200000026.

[2] Karl Beecher, Andrea Capiluppi, and Cornelia Boldyreff. Identifying Exogenous Drivers and Evolutionary Stages in FLOSS Projects. *The Journal of Systems and Software*, 82(5):739–750, May 2009. ISSN 0164–1212. doi:10.1016/j.jss.2008.10.026.

[3] Magiel Bruntink. Towards Base Rates in Software Analytics: Early Results and Challenges from Studying Ohloh. *Science of Computer Programming*, 2013.

[4] Magiel Bruntink. OhlohAnalytics data set and analysis tools. https://github.com/MagielBruntink/OhlohAnalytics, 2013.

[5] Magiel Bruntink. An Initial Quality Analysis of the Ohloh Software Evolution Data. In *Proceedings of the International Workshop on Software Quality and Maintainability, SQM 2014*, volume 65. ECASST, 2014.

[6] Jeffrey C. Carver. Towards Reporting Guidelines for Experimental Replications: A Proposal. In *Proceedings of the 1st International Workshop on Replication in Empirical Software Engineering Research (RESER)*. ACM, May 2010.

[7] Kevin Crowston, Hala Annabi, and James Howison. Defining Open Source Software Project Success. In *Proceedings of the 24th International Conference on Information Systems (ICIS)*, pages 327–340, Seattle, WA, USA, December 2003.

[8] Kevin Crowston, James Howison, and Hala Annabi. Information Systems Success in Free and Open Source Software Development: Theory and Measures. In *Software Process: Improvement and Practice*, volume 11, pages 123–148, 2006.

[9] William H. Delone and Ephraim R. McLean. Information Systems Success: The Quest for the Dependent Variable. *Information Systems Research*, 3(1):60–95, 1992. doi:10.1287/isre.3.1.60.

[10] William H. Delone and Ephraim R. McLean. The Delone and McLean Model of Information System Success: A Ten-Year Update. *Journal of Management of Information Systems*, 19(4): 9–30, April 2003. ISSN 0742-1222.

[11] Sebastian G. Elbaum and John C. Munson. Code Churn: A Measure for Estimating the Impact of Code Change. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 24–, Washington DC, USA, 1998. IEEE Computer Society Press. ISBN 0-8186-8779-7.

[12] Miguel Goulão, Nelson Fonte, Michel Wermelinger, and Fernando Brito e Abreu. Software Evolution Prediction Using Seasonal Time Analysis: A Comparative Study. *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference*, pages 213–222, 2012. ISSN 1534-5351. doi:10.1109/CSMR.2012.30.

[13] Amara Graps. An Introduction to Wavelets. *IEEE Computational Science and Engineering*, 2 (2):50–61, 1995. ISSN 1070-9924. doi:10.1109/99.388960.

[14] Abram Hindle, Michael W. Godfrey, and Richard C. Holt. Mining Recurrent Activities: Fourier Analysis of Change Events. *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference*, pages 295–298, May 2009. doi:10.1109/ICSE-COMPANION.2009.5071005.

[15] Black Duck Software Inc. Ohloh Meta — About Ohloh. http://meta.ohloh.net/us/, 2014.

[16] Siim Karus. Automatic Means of Identifying Evolutionary Events in Software Development. In *Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM)*, pages 412–415. IEEE, September 2013. doi:10.1109/ICSM.2013.60.

[17] Siim Karus. Measuring Software Projects Mayan Style. *Emerging Trends in Software Metrics (WETSoM), 2013 4th International Workshop*, pages 28–34, May 2013. ISSN 2327-0950. doi:10.1109/WETSoM.2013.6619333.

[18] Saiqa Khan and Arun Kulkarni. Reduced Time Complexity for Detection of Copy-Move Forgery Using Discrete Wavelet Transform. *International Journal of Computer Applications*, 6(7):31–36, September 2010. doi:10.5120/1087-1418.

[19] Stefan Koch. Software Evolution in Open-Source Projects - A Large-Scale Investigation. *Journal of Software Maintenance and Evolution: Research and Practice*, 19:361–382, November 2007. doi:10.1002/smr.348.

[20] Meir M. Lehman. Programs, Life Cycles, and Laws of Software Evolution. In *Proceedings of the IEEE*, volume 68, pages 1060–1076. IEEE, September 1980. doi:10.1109/PROC.1980.11805.

[21] Meir M. Lehman. Laws of Software Evolution Revisited. In *Proceedings of the 5th European Workshop on Software Process Technology*, EWSPT '96, pages 108–124. Springer-Verlag, October 1996. ISBN 3-540-61771-X.

[22] A.N. Myna, M.G. Venkateshmurthy, and C.G. Patil. Detection of Region Duplication Forgery in Digital Images Using Wavelets and Log-polar Mapping. In *Proceedings of International Conference of Computational Intelligence and Multimedia Applications 2007*, volume 3, pages 371–377. IEEE, December 2007. doi:10.1109/ICCIMA.2007.271.

[23] Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. Diversity in Software Engineering Research. In *Proceedings of the 9th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 466–476, Saint Petersburg, Russia, August 2013. ACM. ISBN 978-1-4503-2237-9.

[24] Kumiyo Nakakoji, Yasuhiro Yamamoto, Yoshiyuki Nishinaka, Kouichi Kishida, and Yunwen Ye. Evolution Patterns of Open-Source Software Systems and Communities. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 76–85. ACM Press, 2002. doi:10.1145/512035.512055.

[25] Uzma Raja and Marietta J. Tretter. Defining and Evaluating a Measure of Open Source Project Survivability. *IEEE Transactions on Software Engineering*, 38(1):163–174, January 2012. ISSN 0098-5589. doi:10.1109/TSE.2011.39.

[26] Ioannis Samoladas, Lefteris Angelis, and Ioannis Stamelos. Survival Analysis on the Duration of Open Source Projects. *Information and Software Technology*, 52(9):902–922, 2010. ISSN 0950-5849. doi:10.1016/j.infsof.2010.05.001.

[27] Radomir S. Stankoviç and Bogdan J. Falkowski. The Haar Wavelet Transform: Its Status and Achievements. *Computers and Electrical Engineering*, 29(1):25–44, 2003. ISSN 0045-7906. doi:10.1016/S0045-7906(01)00011-8.

[28] Pallavi D. Wadkar and Megha Wankhade. Face Recognition Using Discrete Wavelet Transforms. *International Journal of Advanced Engineering Technology*, 3(1):239–242, January 2012. ISSN 0976-3945.

[29] Jing Wang. Survival Factors for Free Open Source Software Projects: A Multi-stage Perspective. *European Management Journal*, 30(4):352–371, 2012. ISSN 0263-2373. doi:10.1016/j.emj.2012.03.001.

Appendix A

Replication report

This appendix reports on the replication using guidelines proposed in the paper by Carver [6].

About the original study

The original study, "Automatic Means of Identifying Evolutionary Events in Software Development", was conducted by Siim Karus in 2013. The main research question of the original study is:

"Can we use wavelet analysis to find objective warning signs in software projects leading to the end of code evolution?"

Data

Karus used a data set consisting of 27 OSS projects.

18 of the projects were randomly chosen using Google Code Search and contained evolution data from different repositories. The projects vary in team sizes, main programming languages, and project types. 15 of the projects were alive, and 3 had no activity since January 2009 and were considered 'dead'. The evaluation for dead or alive was done in January 2013.

The data set was then complemented with 9 dead projects from SourceForge[1] in order to balance the number of dead and alive projects. Making a total number of 27 projects.

Karus used a specially built tool for mining the software repositories. As a result, complete commit logs of the projects were gathered.

The construction of the project signals was done using the metrics shown in Table X. Karus used two kinds of time-series dimensions: days since first commit, and cumulative code churn. A total number of 20 types of signals were constructed for 27 projects, making a total of 540 signals analysed in the original study.

[1] `http://www.sourceforge.net`

Table X: Project signals in the original study

Time	Frequency	Aggregation
Days since first commit	Active developers	Mean
	Cumulative developers	Sum
	Cumulative LOC added	Sum
	Cumulative LOC churn	Sum
	Cumulative LOC modified	Sum
	Cumulative LOC removed	Sum
	LOC	Mean
	Relative cumulative LOC churn	Mean
	Relative team size	Mean
	Files	Mean
Cumulative LOC churn	Active developers	Mean
	Cumulative developers	Sum
	Cumulative LOC added	Sum
	Cumulative LOC modified	Sum
	Cumulative LOC removed	Sum
	LOC	Mean
	Relative cumulative LOC churn	Mean
	Relative team size	Mean
	Files	Mean
	Commits	Mean

Method

The original study was done in the following steps:

Data preparation. The evolution data of the 27 projects was aggregated along the time-series dimensions. Karus had daily commit data for all 27 projects. The 'days since first commit' metric was aggregated into 7 day-frames (i.e., weekly data), and the time-series of 'cumulative LOC churn' metric was aggregated into 1,000 LOC frames.

The aggregation method for each of the metrics are shown in Table X.

Discrete wavelet transform. The second step was applying the discrete wavelet transform with the Haar filter (Daubechies-2) on the data series. This step resulted in wavelet and filter coefficients for each compression level.

Similar region detection. In the third step, the series of coefficients are compared to find linearly positively similar regions of sub-sequences. In the paper, Karus [16] showed the method for defining what is similar.

The similar sequences occurring at least 3 times are grouped together as a 'pattern'.

Results

Karus [16] found 998 common patterns across different projects. He confirmed that "*... evolution patterns in software projects do express various warning signs leading to or indicating the impending end of code evolution.*".

Karus concluded that: "*... wavelet analysis can be successfully used to detect anomalies* [in software evolution] *before they turn into a long-term issue, allowing action to be taken before it is too late.*",

and, "*In conclusion, we have demonstrated the usefulness for both tracing software evolution and for detecting anomalies in software evolution.*".

About the replication

Level of interaction

During this study, and especially in the beginning, I was in contact with Karus about the implementation of his study. He explained some of the decisions he made and the 'problems' he had conducting the study.

There are many knobs that can be tweaked in this kind of research. Therefore, to be able to fully replicate the results, Karus provided his raw data set and his analysis scripts. The three R scripts he had used for detecting the patterns:

- wavelet analysis: takes the data as input, and outputs the coefficients;
- similar sequence detection: takes the coefficients and outputs pairs of similar regions;
- sequence grouping: takes the pairs of similar regions and outputs the patterns.

Changes

Table XI presents what aspects of the original study were changed in the replication. Each change is compared and motivated.

Table XI: Changes to the original study

Aspect	Original	Replication	Motivation
Data			
Sample size	27 projects	250 projects	A larger data is believed to yield more trustworthy results in terms of the ability to generalise results.
Sample selection	Random and manual complementing	Automatic representative selection	Knowing the representativeness and selecting projects accordingly increases the ability to generalise the results.
Data gathering	Repository mining	Ohloh analysis	The data gathered by the tool [4, 3] was gathered from Ohloh [15]. Ohloh did the source code repository mining and aggregated and analysed this data into monthly facts. Furthermore, the data provided by the OhlohAnalytics tool is already validated and cleansed to be consistent.
Data frames	Daily aggregated to weekly	Monthly	No aggregation of the raw data was done in the replication, because the data provided from Ohloh and OhlohAnalytics is already presented in monthly time frames.
Time series	Days since first commit	Age in months	Similar to the 'Data frames' change, the 'days since first commit' is aggregated in 7 day frames in the original study. Karus aggregates the other metrics as shown in Table X. In the replication this was not the case as the data provided is already in monthly data frames. The aggregation was not done because of issues found with aggregation of LOC as discussed in section 2.2.
Code churn	Sum of LOC added, modified, and removed	Sum of LOC added and removed	Ohloh does not track a modified LOC fact. Therefore, the LOC modified metric could not be added to the calculation of LOC churn.

Analysis			
Signals	$20 \times 27 = 540$ signals (see Table X)	$1 \times 250 = 250$ signals (i.e., LOC)	Only LOC is used because of its simple interpretation and intuitive representation.

Discussion

Time frames

The fact that I used monthly time frames as opposed to weekly time frames, was because of the fact that Ohloh provides no finer grained time frames than monthly. Furthermore, there was no reason to believe that finer grained time frames are necessary for detecting patterns that lead to the end of code evolution for a project.

The 'end of code evolution', or 'project death' is a phenomenon that reveals in several months. In the definition of a dead project (see section 2.3.1), the 'end of code evolution' is the moment that the project died, which in turn is defined as having no code activity for 12 months in a row. Therefore, patterns leading to the end of code evolution will last for months. Weekly data points will probably not add significant value in detecting warning signs.

LOC churn

The 'LOC churn' metric is defined as the sum of lines of code added, modified, and deleted. The data used in the replication of the study did not include LOC modified. Therefore, the LOC churn is calculated as the sum of LOC added and removed. This differs from the original study. However, there was no reason to think of this as a threat to validity because the way it is computed was consistent.

Ohloh inspects differentials when counting lines of code added or deleted. This means that one line modified will be counted as both one line deleted and one line added. When calculating the lines of code churn, the lines that were modified are actually counted twice. Still, it won't introduce problems as the metric of LOC churn is computed consistently across all projects.

Furthermore, in the replication the LOC churn metric is not used as a signal to be analysed, but merely as a measure to indicate any code activity for determining whether a project is dead.

Signals

Karus processed 540 signals in total, as opposed to 250 signals in the replication. The quantity of signals is not related to the quality of the patterns it is able to detect, because the signals are processed and analysed independently (i.e., not mixed).

The choice of the signal should depend on the relation between the time and frequency domains. For time and 'lines of code' (LOC), this relation can be interpreted with some intuition, which eases validation of the results.

Evaluation

The results from the replication are comparable to the results of the original study. Both studies found patterns in software evolution by using wavelet analysis. Both studies found patterns that may lead to the end of code evolution.

The difference in results between the original and the replication can be found in the evaluation of the patterns and the evaluation of wavelet analysis as a tool to find objective warning signs.

Wavelet analysis works best on continuous data series (section 2.2). Furthermore, the ability of wavelet analysis to find patterns depends on the data at hand (chapter 7).

Some patterns show strong characteristics of warning signs, such as patterns of type A that were detected near the end of evolution of dead projects. As a result of survival analysis these type A patterns suggest to be a sign of warning for the death of a project.