

Automatic Means of Detecting Warning Signs in Software Evolution

ing. Martijn Endenburg

martijn.endenburg@gmail.com

June 11, 2014, 43 pages

Supervisors: dr. Magiel Bruntink (UvA), ir. Rutger Pannekoek (VNU)

Host organisation: VNU Vacature Media, De Persgroep Nederland

Publication status: For review



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

© 2014 Martijn Endenburg

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. Conducted at VNU Vacature Media, De Persgroep Nederland, Mt. Lincolnweg 40, 1033 SN, Amsterdam.

Revision history

Version	Recipients	Remarks
20140324	ir. Rutger Pannekoek	draft Introduction, Method
20140410	dr. Magiel Bruntink, ir. Rutger Pannekoek	draft Introduction, Method, Research
20140611	dr. Magiel Bruntink, ir. Rutger Pannekoek	for review

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 to use, or whether 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

Contents

Abstract	iv
List of Figures	3
List of Tables	4
1 Introduction	5
1.1 Problem statement	5
1.2 Motivation	5
1.3 Research question	6
1.4 Outline	7
2 Research method	8
2.1 Research approach	8
2.2 Hypotheses	8
2.3 Validation	9
2.3.1 Dead projects	9
2.3.2 Warning signs	9
3 Background	10
3.1 The OSS organisation	10
3.2 Software evolution	10
3.2.1 Feedback loops	10
3.2.2 Maintaining complexity	11
3.2.3 Activity	11
3.3 Success factors	11
3.3.1 Indicators of success	12
3.4 Evolution in OSS	14
3.4.1 Project transition	14
3.4.2 Project characteristics	14
3.4.3 Software evolution prediction	15
3.5 Project survivability	15
3.5.1 Project causes of death	17
3.6 Wavelet analysis	18
3.6.1 Waveforms and wavelets	18
3.6.2 Discrete wavelet transform	18
3.6.3 Wavelet functions	20
3.6.4 Haar wavelet	20
3.6.5 Discrete Haar transform	21
4 Research	23
4.1 Data selection	23
4.1.1 Data gathering	23
4.1.2 Data validation	23
4.1.3 Dead projects	25

4.2	Wavelet transform and analysis	25
4.2.1	Project signals	25
4.2.2	Wavelet transform	26
4.2.3	Wavelet analysis	26
4.3	Pattern classification	27
5	Results	28
5.1	Data	28
5.2	Dead projects	28
5.3	Sequences and patterns	29
5.4	Pattern classification	31
6	Analysis and Conclusions	32
6.1	Evaluation	32
6.1.1	Wavelet analysis	32
6.1.2	Patterns	33
6.1.3	Survivability	33
6.2	Conclusions	34
6.3	Threats to validity	35
6.4	Future work	35
7	Replication report	37
7.1	About the original study	37
7.1.1	Data	37
7.1.2	Method	38
7.1.3	Results	38
7.2	About the replication	39
7.2.1	Level of interaction	39
7.2.2	Changes	39
7.2.3	Discussion	40
7.3	Evaluation	40
	Bibliography	41

List of Figures

I	The Delone & McLean model of information system success (1992)	12
II	Example of an audio wave	19
III	Shifting/translating a wavelet	19
IV	Filtering/dilating a wavelet	20
V	The Haar functions	21
VI	Discrete wavelet transform using Haar filter	22
VII	LOC signal of project #19012	26
VIII	Some patterns found during wavelet analysis	30
IX	Kaplan-Meier estimation of the survival function of projects regarding type A patterns	34

List of Tables

I	Monthly data fields	24
II	Dead projects' final age (in months)	28
III	Patterns detected and occurring	31
IV	Pattern types	31
V	Project signals in the original study	38
VI	Changes to the original study	39

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 organisation using 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, as well as the hypotheses. 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, as well as drawing conclusions and answering the research questions and hypotheses. Finally, chapter 7 compares this replication to the original study.

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.

Further evaluation of the data should be done prior to the selection of the data set for the study. It is expected that not all data is suitable for analysis.

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.

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.

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 Hypotheses

In the search for patterns that may lead to the end of code evolution, the following hypotheses are defined. During the research, these hypotheses will be either confirmed or refuted.

H1 The analysis of waveforms will need a subsequent data series in order to function properly.

An investigation will be made to determine the differences between analysing subsequent data series and data series containing gaps. Whether this hypothesis is true will help answering Q3.

H2 Projects with a pattern occurring at the end of code evolution of a dead project have a higher chance of dying.

The patterns occurring at the end of code evolution in dead projects are expected to be more likely a warning sign. A survival analysis of projects having these kinds of patterns will be conducted to confirm or refute this hypothesis.

2.3 Validation

In order to validate the patterns 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.

2.3.1 Dead projects

A project is considered 'dead' if it complies to the following properties:

1. the project showed no change in 'lines of code' (LOC) for the last 12 months;
2. the project has had 0 contributors in code for the last 12 months;
3. the project has had 0 LOC churn¹ for the last 12 months.

The above properties combined specify that the project has had no code activity for the past year, meaning the code evolution of the project has stopped. 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 (section 2.3.1) are still dead beyond 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 to verify the properties of a 'dead' project.

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 at 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 analysis of survival will be done to evaluate this (H2).

¹LOC 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.

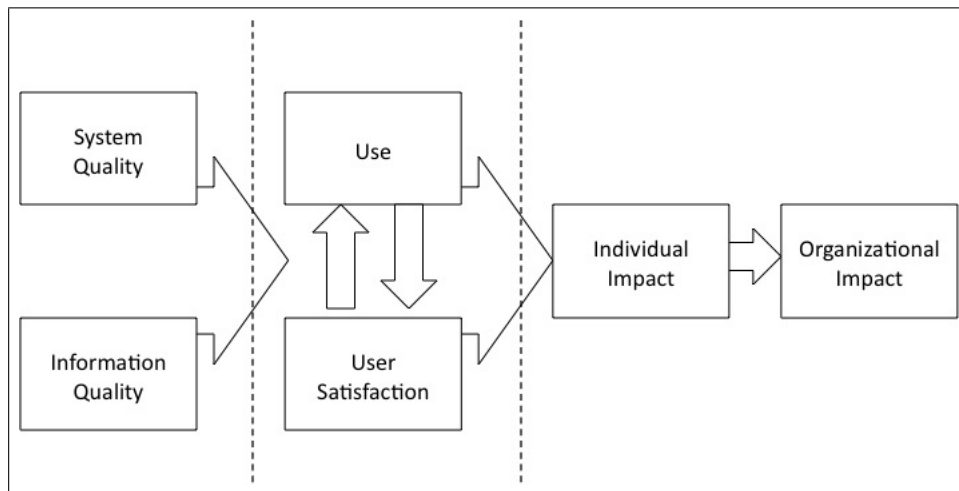
The number of downloads is an indicator for success; a frequently downloaded project is likely to be successful.

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 SourceForge¹. 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 ARIMA² 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].

²auto-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.

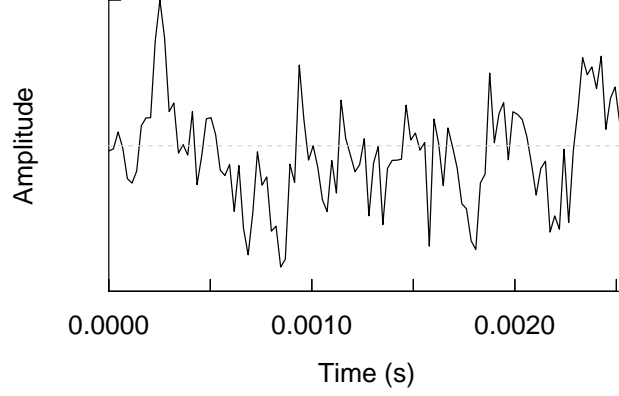
In visualisations of waveforms, the time domain is often plotted on the horizontal axis, and the frequency domain on the vertical axis. In the example of an audio wave (see Figure II), the frequency domain may represent the amplitude, or the frequency, and the time the duration of the amplitude or frequency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

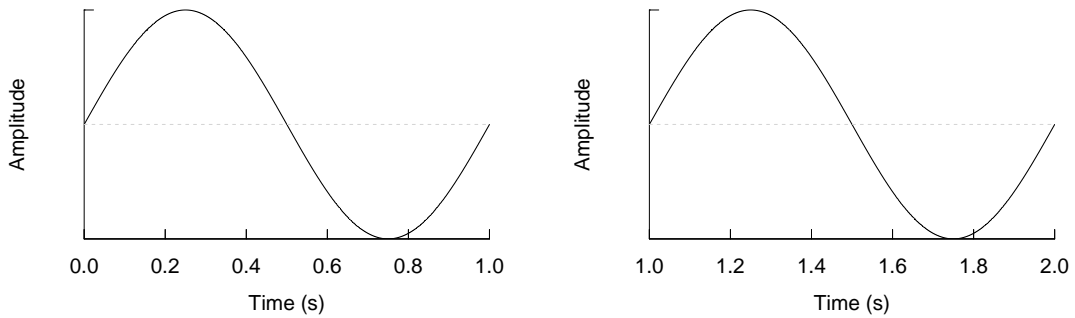
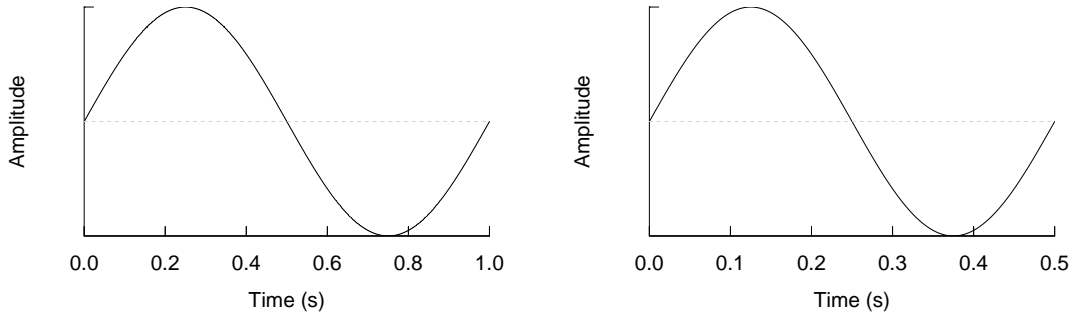


Figure IV: Filtering/dilating a wavelet

(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)

$$\Psi : T \rightarrow F$$

$$\Psi(t) = f, \text{ such that } t \in T \text{ and } f \in F.$$

A bijective function mapping T onto F and producing the shape of the wavelet.

Scaling function (father wavelet)

$$\Phi : F \rightarrow T$$

$$\Phi(f) = t, \text{ such that } t \in T \text{ and } f \in F.$$

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. All Daubechies wavelets can be defined by the scaling filter.

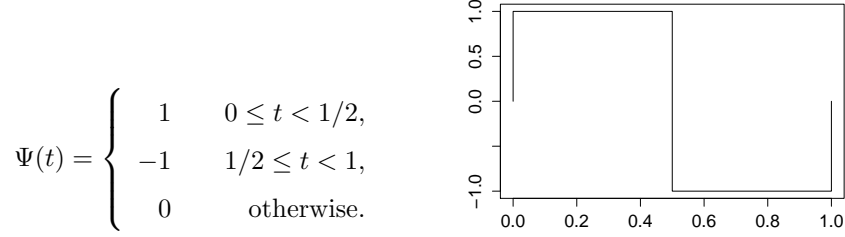
3.6.4 Haar wavelet

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)



(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 compact³, dyadic⁴, orthonormal⁵ 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).

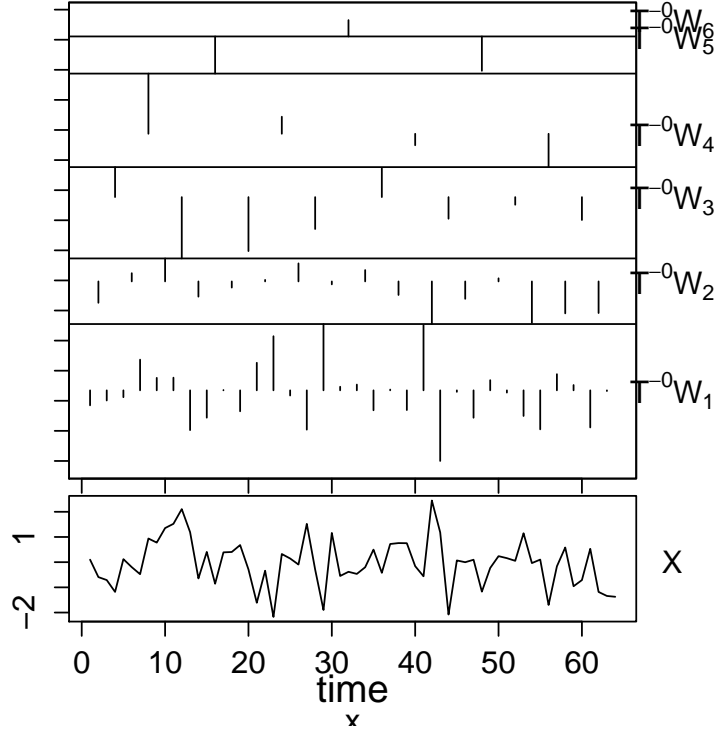
³Compact support: the ability of a function to be scaled smoothly.

⁴A dyadic function scales in powers of 2.

⁵Orthonormal: domain and range are orthogonal and of equal length.

Figure VI: Discrete wavelet transform using Haar filter

W_1, \dots, W_6 represent the levels of decomposition in reversed order, the lower most graph shows the original signal. The coefficients plotted at W_6 (top most graph) are generated in the first step of decomposing. Each level from W_6 to W_1 adds detail from the original signal. The highest level (W_1) 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 of four months.

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 Ohloh¹: “*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 by Bruntink [4] 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 are 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

Subsequent data 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	<i>loc_added_fact, loc_deleted_fact</i>	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	<i>loc_fact, abs_loc_growth</i>	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	<i>month_fact, year_fact</i>	The age of the project in months measured since the first data point; starts at 0.
age_in_years	<i>age_in_months</i>	The age of the project in years measured since the first data point; starts at 0.
cumulative_commits_fact	<i>commits_fact</i>	The total number of commits since the first data point (i.e., where <i>age_in_months</i> = 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 <i>updated_at</i> field.
year_fact	Raw	The year value of current month's analysis. Extracted from <i>updated_at</i> 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. It is expected by Hypothesis H1 that a subsequent series of data is required to be able to analyse time series for a project. 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

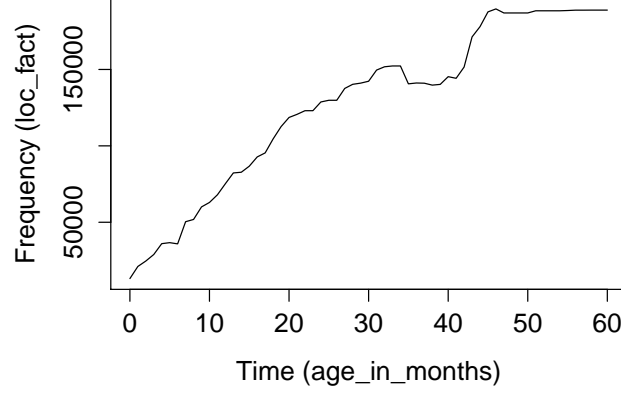
In the search to find patterns that are warning signs leading to the end of code evolution, the projects that satisfy the definition of a dead project (see also section 2.3.1) were extracted from the 250 projects.

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 rationale for choosing '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 same metric as in the original study by Karus [16] is used. The *LOC* metric is constructed as the sum of *loc_fact*, *comments_fact*, and *blanks_fact*.

For the wavelet transform and analysis, I have used 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 resulted from the wavelet transform are analysed to find *similar sequences*. A sequence is defined as a subsequent 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 sequences of other projects. 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. The process of finding similar sequences constructs ordered pairs of sequences from one project to the other. The first element of the pair is the project of which the sequence was first discovered, the second is the project of the similar sequence. Similar sequences may be found within one project, but can also be found across projects.

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.

The sequences that are instances of the same pattern are grouped together and added meta data, such as the number of occurrences, the list of projects it occurs in, and whether this list of projects contains dead, alive, or both dead and alive projects.

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

When *detecting* a pattern in project A , it means it was encountered by analysing that particular project. Finding an *occurrence* of a pattern means that the pattern was found by evaluating other project B 's sequences against those from project A . A pattern is then recorded as a tuple of a sequence detected in project A and a set of at least 2 other sequences being similar, occurring elsewhere.

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, different types of patterns for dead projects are introduced:

Type A. Type A patterns strictly detected at the end of the evolution data of a dead project. More formally, if a dead project P starts at point x and ends at point y , the pattern starts at x' and ends at y .

Type B. Type B patterns strictly *not* detected at the end of evolution data of a dead project. Such patterns may occur anywhere else in the lifetime of a dead project.

Type AB. Type AB patterns are patterns that weakly satisfy the properties of both type A and B patterns. This category is needed to distinguish strict from weak patterns of type A.

The above types partition the total set of encountered patterns into three disjunctive sets.

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

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 by 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.

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 in 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 II.

Table II: 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 II 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. For 9 of the projects activity is very low or rapidly decreasing; 5 projects were migrated to another source code repository; and for 3 projects the activity is increasing after a long period of no activity.

The 9 projects with very low or decreasing activity show signs of 'dying'. 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.

The 5 migrated projects for which the tracking information is not updated at Ohloh 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, 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.

The similar sequence identification found 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.

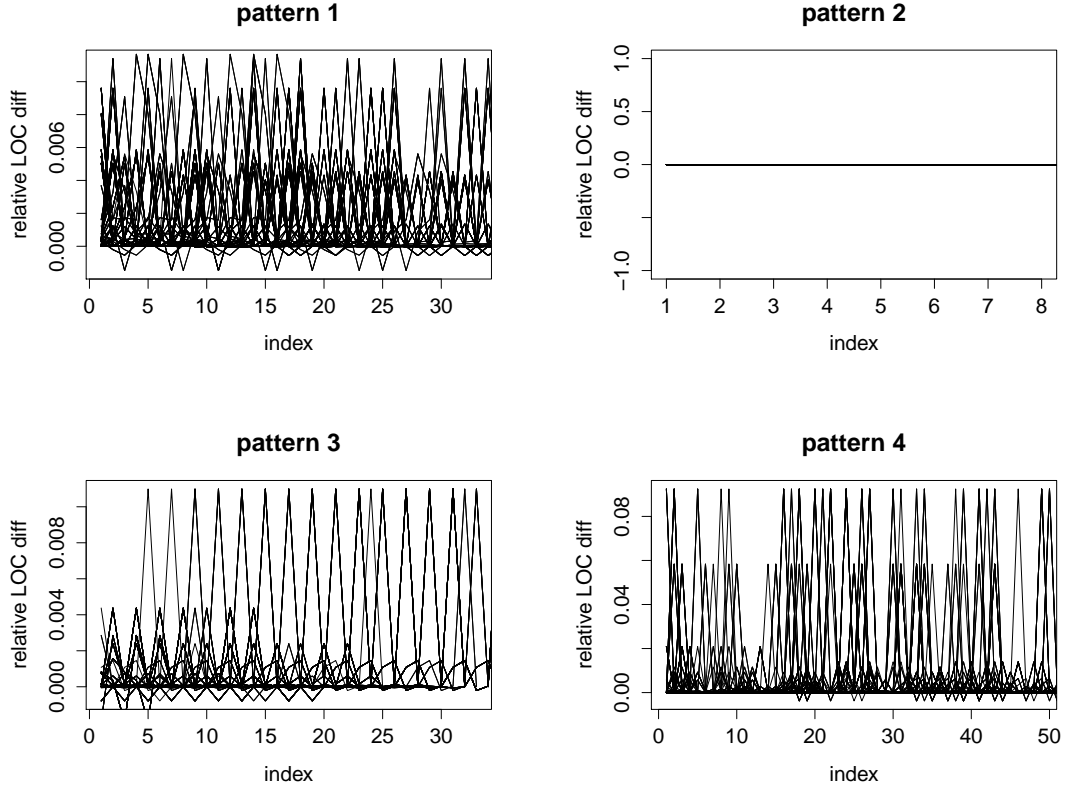
The second wavelet analysis step aggregated similar sequences across multiple pairs together to form patterns. The detection of these 'popular sequences' found 16,049 patterns. The patterns consist only of sequences of scale/filter coefficients.

Some patterns that were found are shown in Figure VIII. The four graphs are plots of patterns recomposed onto the original signal, however, transformed to *relative LOC change* to make them more comparable in the graph.

The graphs all have multiple (> 140) occurrences of the pattern plotted of one or more projects in the same graph. The graphs show that the sequences composing a pattern have similar waveforms. Each graph contains a multiple of plotted sequences as stated in the figure.

Figure VIII: Some patterns found during wavelet analysis

(pattern 1) 216 sequences;
 (pattern 2) 156 sequences;
 (pattern 3) 151 sequences;
 (pattern 4) 141 sequences.



Pattern no. 2 illustrates a pattern indicating no change in LOC for 8 months.

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 3th level, the average pattern length is equal to 4 coefficients. Every subsequent level doubles the average pattern length. This is expected as there are twice as many coefficients available. This means the same pattern is detected at multiple levels. Additionally, it can be said that a pattern has multiple levels.

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

Table III: 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	
total			111,848	100.00%
in dead projects only			111	0.10%
in both dead and alive projects			111,737	99.90%
Patterns detected in alive projects			Occurrences	
total			1,560,883	100.00%
in alive projects only			37,333	2.39%
in both dead and alive projects			1,523,550	97.61%

As can be seen from Table III, 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.

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 this group has 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 IV. For each subset, it is stated in how many projects a pattern of that type was found.

Table IV: 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%						

The patterns of all three groups were detected in dead projects, but may occur in both dead and alive projects.

In search of patterns that lead to the end of code evolution, patterns of type A will be most interesting, because these patterns last until the end of evolution data of dead projects.

The evaluation of type A patterns is described in section 6.1.3.

Chapter 6

Analysis and Conclusions

6.1 Evaluation

6.1.1 Wavelet analysis

In verification of Hypothesis H1, the requirement of a subsequent data 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 LOC value:

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

In signal S the two pairs having first entry 3 and 4 are missing. The implementation of discrete wavelet transform in the 'wavelets' R package requires an indexed series as input wavelet. 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 was done on how to fix such gaps. The use of four different methods of 'guessing' the data was explored and the results were 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 with same values between point 2 and 5, which shows a flat line when plotted in a graph.

Filling the gaps with *mean* values, will also result in a signal with same values. It is similar to last observation carried but using the mean is excluding the two known observations, whereas last observation carried is inclusive. Using mean, the number of second entries having same values will therefore be smaller than using the last observation.

The *piecewise constant* interpolation results in a signal having two smaller flat lines, because the repeating values are 'snapped' to the closest neighbour.

Finally, the *linear interpolation* will result in a signal that linearly flows towards the next known value. The advantage of this technique is that there will not be a flat line in the signal, but the disadvantage is that the numbers are new in the signal, possibly 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 LOC during that period. This is undesired as 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 LOC, 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, adding data will add noise. However, in general having gaps in the data and close the gaps by adding data negatively influences the reliability of detecting patterns. Therefore, Hypothesis H1 is confirmed.

6.1.2 Patterns

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.

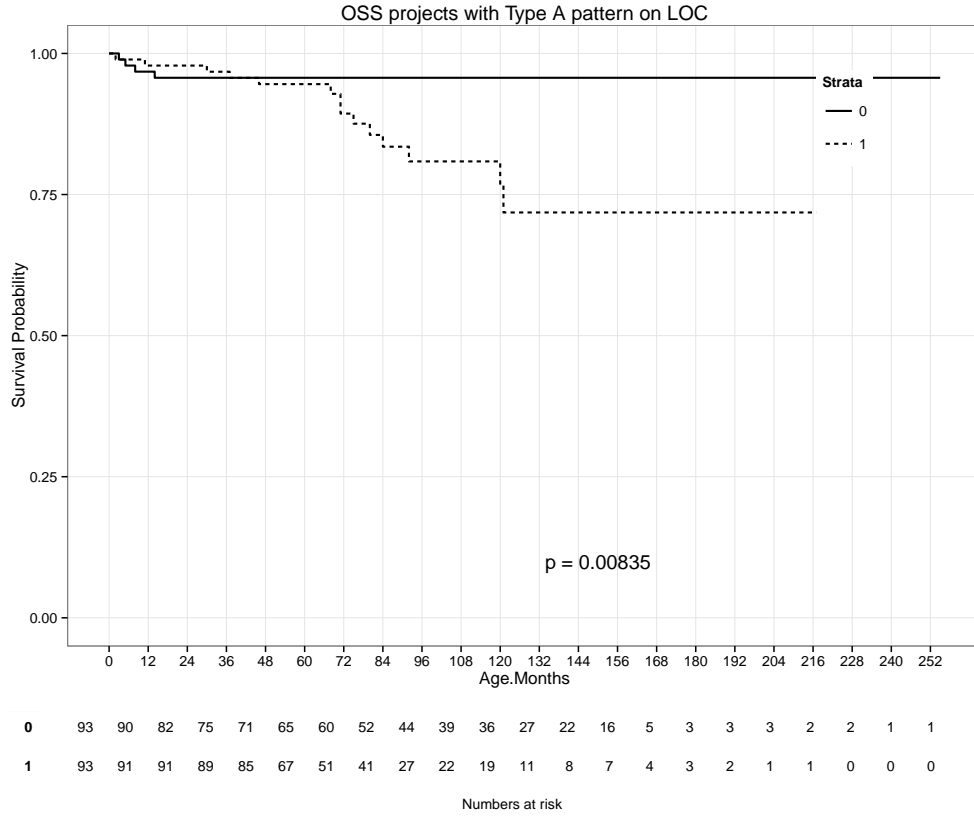
The patterns that were detected show strong similarity. The similarity was demonstrated in Figure VIII in section 5.3. The figure presented four graphs of four distinct patterns plotted as wavelets of the original signals. Each graph contains between 141 and 216 wavelets. Because of the similarity, the shape of the pattern is still quite clear.

6.1.3 Survivability

The type A patterns would be the best candidates for being 'warning signs' as they are detected at the end of evolution in dead projects. To determine whether the chances for a project to die increase when having a type A pattern, two groups are made. One group consisting of projects having the pattern, and a second group of equal size consisting of projects not having the pattern.

Both groups of 93 projects contain alive and dead projects. The group with projects without the pattern is representative to the data set as a whole. Selecting projects and scoring for representativeness is done with the tool by Nagappan *et al.* [23]. For each project of both groups it is recorded at what age the project died, that is, if it died. For the others, the maximum age in the data set is used. In Figure IX the Kaplan-Meier estimation of the survival function of the projects is shown for these groups.

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



The Kaplan-Meier estimation of survival function of the projects as depicted in Figure IX suggests that projects in group 1 - the projects having the type A pattern - die earlier than projects in group 0 - the projects without the pattern. This suggests that having found a type A pattern in a project shortens the project's life. Therefore, Hypothesis H2 is confirmed; projects with a pattern occurring at the end of evolution of a dead project (i.e., type A pattern) have a higher chance of dying.

6.2 Conclusions

In this study, the patterns detected 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. The type A patterns all show a stagnation of LOC changes, that is, very few changes relative to the project size.

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 [Q2, Q3].

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. In section 6.1.1 the confirmation of Hypothesis H1 is argued.

In the case an input LOC signal for analysis is not continuous within the start and end boundaries of the signal, the wavelet analysis will detect patterns that are not trustworthy. Wavelet analysis may find more false-positive evolutionary events [Q3].

In this study, I have shown that wavelet analysis is able to find patterns that increase the chances of a project to end. As discussed in section 6.1.3, it appears that there is a relation between a pattern of type A and the death of a project, however, it is not found to be a causal relation.

Furthermore, as the patterns found during analysis are subject to the data set as a whole, I cannot conclude that wavelet analysis is able to find *objective* warning signs in OSS projects [Q1].

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, but to make it consistent. 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 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 metrics could be team size, developer's churn (the number of developers added and removed from the team), contributor's code activity, bug reports, defects per kLOC etc.

Chapter 7

Replication report

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

7.1 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?”

7.1.1 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 SourceForge¹ 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 V. 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 V: 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

7.1.2 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 V.

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'.

7.1.3 Results

Karus 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.*" [16].

Karus [16] 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.*”.

7.2 About the replication

7.2.1 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.

7.2.2 Changes

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

Table VI: 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 V. 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 6.1.1.
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 V)	$1 \times 250 = 250$ signals (i.e., LOC)	Only LOC is used because of its simple interpretation and intuitive representation.

7.2.3 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.

7.3 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 subsequent series of data (section 6.1.1). Furthermore, the ability of wavelet analysis to find patterns depends on the data at hand (section 6.2).

Some patterns show strong characteristics, such as patterns of type A that were detected at 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. However, it cannot be concluded that wavelet analysis is capable of finding warning signs objectively, as the sequences making a pattern strongly relate.

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:10.1561/02000000026.
- [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. Stanković 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.