University of Potsdam

Master's Thesis

Eliciting Expertise based on Time Series Analyses of Code Complexity Metrics

Bestimmung von Expertise basierend auf Zeitreihenanalysen von Komplexitätsmetriken

Author:
Philipp Giese

Supervisors:
Prof. Dr. h.c. Hasso Plattner
Dr. Matthias Uflacker
Ralf Teusner

A thesis submitted in fulfilment of the requirements for the degree of Master of Science

in the

Enterprise Platform and Integration Concepts Group
Hasso Plattner Institute
28.03.2014

Declaration of Authorship

I, Philipp Giese, declare that this thesis titled, "Eliciting Expertise based on Time Series Analyses of Code Complexity Metrics" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

SIGNED: _____ DATE: _____

Abstract

Eliciting Expertise based on Time Series Analyses of Code Complexity Metrics

by Philipp Giese

The number of software systems that surround us in our everyday life is rapidly growing. In order to keep these systems practicable, new features have to be implemented resulting in a growing code base. While senior developers might be able to oversee a large software ecosystem, it is difficult for new developers to get acquainted with those extensive amounts of code. Even worse, losing a senior developer might cause a severe knowledge gap within the organisation, which requires new developers to be trained. This results in serious financial losses and the chance of defective software.

To detect and prevent such problems, we implemented an open source framework which, contrary to other metric based approaches, utilises time series analysis of complexity measures. We monitor code complexity using metrics such as the McCabe complexity or the Halstead metrics. The results have been evaluated using three different use cases and were verified with a survey at a successful German startup. Our framework is capable of determining the expertise of developers which we use to recommend experts, who can offer help and guidance. Organisations are enabled to detect the parts of their software where developers lack expertise and take preventive actions to keep the collective code knowledge at a high level. Additionally, we are able to classify frontend and backend developers and also find those developers, who can work in both areas, which facilitates organisations to make better staffing decisions and increase the overall performance of their development teams.

Kurzzusammenfassung

Bestimmung von Expertise basierend auf Zeitreihenanalysen von Komplexitätsmetriken

von Philipp Giese

Im täglichen Leben umgibt uns eine steigende Vielzahl von Softwaresystemen. Damit diese Systeme benutzbar bleiben, werden neue Funktionen hinzugefügt, was zu einer stetig wachsenden Codebasis führt. Während erfahrene Entwickler in der Lage sind, solche großen Softwarelandschaften zu überblicken, gestaltet sich dies auf Grund der großen Menge an Code für neue Entwickler schwierig. Noch schwieriger wird es, wenn Entwickler die Organisation verlassen, was erfordert, dass neue Entwickler angelernt werden müssen. Dies führt im Allgemeinen zu erheblichen finanziellen Einbußen und einer gesteigerten Gefahr von Fehlern in der Software.

Um solche Probleme frühzeitig zu erkennen und vorzubeugen wurde im Rahmen dieser Arbeit ein Open Source Framework entwickelt, welches im Gegensatz zu bestehenden Ansätzen Zeitreihenanalysen von Komplexitätsmetriken nutzt. Als Grundlage für unsere Schlussfolgerungen werden Komplexitätsmetriken wie die McCabe Komplexität oder die Halstead Metriken herangezogen. Die Messungen wurden im Rahmen von drei Fallstudien evaluiert und mit einer Studie bei einem erfolgreichen Berliner Startup verifiziert. Das entwickelte System ist in der Lage, die Expertise von Entwicklern automatisch zu bestimmen und kann dadurch Experten vorschlagen, die bei Fragen und Problemen qualifizierte Ratschläge anbieten können. Unternehmen können Stellen in ihren Softwaresystemen aufdecken in denen ihre Entwickler nur einen geringen Überblick besitzen und so rechtzeitig Maßnahmen einleiten um die umfassende Code Übersicht sicherzustellen. Zusätzlich können Entwickler hinsichtlich ihres primären Arbeitsbereiches unterschieden werden. Eine Einteilung in Frontend-, Backend- und universell einsetzbare Entwickler ermöglicht es Organisationen bessere Entscheidungen bezüglich des Einsatzes ihres Entwicklungsteams zu treffen und so die Gesamteffizienz zu erhöhen.

Acknowledgements

So, here it is: My personal Oscar moment. The one part of this thesis where I don't have to be too careful concerning the form or scientific method. I have always found it funny that, even though there is clearly only one name under "Author" on the front page of the book in your hands, it is kind of common sense not to refer to oneself as "I", but "we". People have told me to do this in order to express that even a Master's Thesis is never the work of simply just you. And, though I really thought: "Yeah, but it's clearly me who works through the weekends and stays up late into the night", I have now realised that it's true. I would not have been able to write this thesis without the support of *a lot* of other people. Simply thanking them in the next few sentences hardly makes up for the time, money, and nerves they have invested in me. For those among them who want a little bit more I came up with an idea. As at least I haven't yet given up on the dream of becoming famous one time, here is what you get. You find yourself among those special few who are written down in the work which enabled this great person to start his life and which has lead to the glory that now surrounds him.

You are welcome.

We shall now get to the point where I shout out to all the folks who made this thesis possible. I have put much thought into whom to name first and whom last and I think it should be clear that if your name is in my thesis, you *are* important. However, I had to make a decision and, sticking to the Hollywood metaphor, you can imagine this part to be the credits which sum up the main characters in my life so far. The actors are listed *in order of appearance.*

First of all I want to thank my mother as she is not only a crucial part of the mere fact that I am able to write this, but also because she has unconditionally supported me throughout my life. Unfortunately she passed away way too early and therefore cannot read this anymore which makes me unendingly sad. That brings me to my father who now has to make up for both of them and does a great job doing so. To him I say: "This thesis is proof, that I have now reached what I always wanted, and what both of you have always wished for". So dad, thanks for being there. Even though you do not tell me how you feel all the time, I never felt like you didn't believe in me and I definitively always know when you are proud of me. Thank you. I also want to thank my uncle who introduced me to computers altogether which is kind of a big deal, if you graduate in software engineering.

I want to thank Julian and Katja, who have been, and still are, my best friends. Both of them are joined by Paul, Peter, Daniel, and Micha whom I met during my years at the university. Without you guys all those years would have been only half the fun – and I am clearly making an understatement here.

Nata, you knew you would end up in your own paragraph, didn't you? You are the best non-girlfriend I have ever had and you clearly know how to bring me back to earth when my head is too high up in the clouds. Thank you for that, and being there when I needed you, unconditionally.

How I work and what I do would probably not be the same, if I would not have done that internship at Signavio. I had no idea that, five years later, I would still work at the company, use it in my Master's Thesis, and still enjoy going to the office. A management guy of a large German company once told me: "If you don't wake up on monday morning and think: 'Thank God the weekend is over, I can finally head back to work!' ", then you should probably get another job. Well, I don't have that *exact* thought when I get up on mondays, but I still enjoy working at Signavio a lot and I am happy to continue to do so in the future. Thank you, Willi, Nico, Gero, Torben, Stefan, Sven, Gerrit, and all the other folks who make Signavio such an awesome company.

I'm now going to make a bold move and also thank my girlfriend Franziska for making a bold move herself and being with me. I hope that, given some years time, we can take this dusty book off the shelf, open it, and read these words once more – together.

Last, but definitively not least (not least at all), I want to thank my supervisors at the university. Ralf and Matthias, without your feedback it would have been even harder to find the right course for this thesis. I sincerely hope that the result satisfies you as much as it satisfies me.

I hope all of you, who have made the effort and actually read all the stuff above, have noticed that I never used the phrase "I have to thank..." because I wanted to emphasise, that I really *want* to thank all of you. The gratitude I feel is true and so are my acknowledgements.

Thank You.

Contents

For my mother.

Chapter 1

Introduction

> *Scientific knowledge is an enabling power to do either good or bad – but it does not carry instructions on how to use it.*
>
> – Richard Phillips Feynman

In every part of science measurements form a crucial source of information [1]. In the field of computer science or software engineering researchers are split into two camps: Those who claim that software can be measured, and those, who argue that software cannot be analysed or measured. Static code analysis has been around since the early nineteen-seventies [2] and research regarding this topic is still underway. We believe the ability to take measurements is a crucial part of every science and would like to quote Lord Kelvin, a physicist who lived from 1824 – 1904 and said that: "When you can measure what you are speaking about, and express it into numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind: It may be the beginning of knowledge, but you have scarcely in your thoughts advanced to the stage of science."[3]

Software measurements, or software metrics should be seen as a means of help for developers. Nonetheless developers today often times mistake them as a threat. Managers who rely solely on software metrics in order to assess the effectiveness of a developer or the quality of the written code are not understanding them correctly. Wiegers [4] states that "Metrics data is intrinsically neither virtuous nor evil, simply informative. Using metrics to motivate rather than to learn has the potential of leading dysfunctional behaviour, in which the results obtained are not consistent with the goals intended by the motivator." According to [1] software metrics should rather be used in a way so that they form:

1. A basis for estimates,
2. To track project progress,
3. To determine (relative) complexity,
4. To help us to understand when we have achieved a desired state of quality,
5. To analyse the defects,
6. And to experimentally validate best practices.

To us the first point is the most important one. As fast as software evolves nowadays, a metric can only be used as an estimate for something.

However, software metrics can help detect common pitfalls during the software development process a compiler might miss. For instance in the aviation sector software metrics are used in order to verify if the code which is produced by the developers holds under a set of given rules [5]. We want to mention that analysis should not only be run after the code is used in production. Measuring certain code metrics while the development is underway can yield imminent improvements. For example one of the most popular code metrics introduced by Thomas J. McCabe in 1976 [6] has found it's way into most popular code metric tools. It uses aspects of graph theory and applies them to source code in order to measure the complexity of code while it is written and assists the developer to write code which is easier to understand. This is achieved by reducing the complexity of each unit (i.e. method or function). We argue that any piece of code is easier to maintain the less complex it is. Another advantage is that in the majority of cases code which is less complex requires fewer tests [7]. Measuring metrics can therefore save time as it can reduce the number of tests which need to be written and therefore fits perfectly into a development cycle that includes automated testing [2].

In the following chapters we will show how software metrics such as McCabes cyclomatic complexity and the measures introduced by Halstead in 1977 [8] can be used in order to gain insights on how developers improve over time.

This thesis should not be used as a means to assess the performance of individuals. In every software project exist parts which can be considered very good code, but would produce poor values if a measure is applied. Judging only from these values will rather decrease the performance of an individual developer [9]. What we intended during the research that has lead to this writing was to create a tool which can be used to present developers unbiased information on how they write code. This way we might help them to improve, and show them the importance of watching out for complexity. As for the manager we argue that metrics can be seen as guidelines to make better decisions and

better plan and schedule activities [10]. Even with little to no technical knowledge a manager could identify which parts of the code base need more attention and channel resources respectively, reducing the risk of failures in the product [11]. In any way he or she must always keep in mind that metrics can only be seen as a hint and that decisions can only be made with proper guidance.

1.1 Contribution

In this thesis we will present a framework which in great parts does not rely on any specific programming language. We argue that complexity can be defined on a higher level using more abstract measures which do not require constructs which are only available in a certain set of languages. However, we also note that due to the vast amount of programming languages we cannot assure that any language will be compatible to our approach. Furthermore, besides measuring complexity we also use metrics that are concerned with coupling and information flow when they can be applied. This decision is based on the assumption that software which is partitioned into reasonable modules, that are then orchestrated in order to fullfill the goal of the software will be easier to maintain and understand than software which only consists of one large component [12].

We will not examine any mathematical model which might be the basis for certain software, nor will we draw any conclusions from how code is syntactically structured and what might be considered a code smell [13–15].

Using our approach, one can outline points of interest inside a larger code base. Such points of interest can be:

- Places with high complexity which might be hard to maintain and are likely to cause problems.
- Places with moderate or low complexity that can be used as examples to teach other developers.
- Places where only a few developers have expertise and which might be hard to maintain if the developers who are mainly concerned with the code leave the organisation.

To better grasp what we try to achieve, we formulate the following research questions.

1.2 Research Questions

Q1: Can developers be classified to be either frontend or backend developers?

In almost any software project there are mainly two groups of developers. One group that is concerned with the overall system logic and also how data is handled and transformed in order for the software to accomplish its purpose. The other group's main concern is to present the results in a user interface. We want to show whether or not it is feasible for an automated system to distinguish between both groups of developers up to a certain amount of certainty.

Q2: Can we effectively determine the main contributors of a software project?

During the life-cycle of every software project the developers who write the software change from time to time. New developers are hired or join the project and others leave it. We want to examine if its possible to accurately find out which developers are the main contributors of a project at any given time.

Q3: Does the code of developers improve over time according to software metrics?
We argue that the quality of the code a developer writes will increase the longer a developer participates in a software project. Any developer should strive to improve and refactor code that he or she encounters and considers to be of poor quality. Quality in this case will be determined using different code metrics. Ultimately, if every developer in a software project constantly improves the code quality of the files he or she is working on then also the quality of the whole project should improve.

Q4: Can experts for parts of the software be determined automatically?
If we can automatically determine (1) which developers are responsible for which part of the software, (2) who are the main contributors, and (3) how a developer has improved over time we argue that we can draw conclusions concerning expertise inside a software project. We assert that developers that constantly improve code and also contribute a lot to a software project can be considered experts for that respective part of the software to which they contribute the most. A system which supports such kind of analysis can help developers find the right individuals to approach when they are stuck with a problem. Having such information at hand could help new developers to solve problems they encounter faster and easier.

1.3 Thesis Outline

We will first give an introduction into the matter of measuring the complexity of source code in Chapter 2. Subsequently in Chapter 3, we will present the framework we implemented in order to measure source code. The results produced by our framework will be evaluated using three different use cases which we introduce in Chapter 4. In Chapter 5 we present related work regarding the topics measuring source code, aggregating the results in order to draw conclusions on a higher level, and eliciting insights regarding the expertise of developers. We present and evaluate our findings in Chapter 6 and highlight future work in Chapter 7. Finally, we we conclude in Chapter 8.

Chapter 2

Measuring Complexity

Life is really simple, but we insist on making it complicated.

– Confucius

In the following chapter we will present the code metrics that we employed in order to perform the static code analysis. A lot of work of researchers all over the globe has gone into the subject of code metrics and how they can be used to gain valuable insights into a code base. We will first show the relevance of code metrics in general in Section 2.1 and then give a brief introduction into the metrics we found especially applicable for the given use case. Already at this point it is important to understand that code metrics have to be chosen on a case to case basis. No set of metrics will fit all given use cases and thus this evaluation has to take place every time code metrics should be used to quantify aspects related to source code. Each metric we use has been chosen not only based on a diligent literature review [5, 9, 10, 16–21], but also using the evaluation framework described in [22]. For each metric in that pool we evaluated the purpose, scope, and the attributes that need to be measured. We then only chose those metrics which seem to be statistically independent judging from what they measure, how they measure it, and which variables are used in order to compute their result. In our case, these are the McCabe complexity (c.f. Section 2.2), a subset of the Halstead metrics (c.f. Section 2.3), and affarent and efferent coupling (c.f. Section 2.4).

2.1 Evaluation of Code Metrics

As code metrics cannot be used as a means to assess whether a program is working properly or not or if the code produced by a developer is free of any errors, a lot of

organisations do not include the measurement of metrics into their development cycle at all. While this is true, code metrics can still be used in order to find parts of a software which are likely to cause trouble in the future. Also, if developers have a basic understanding of the metrics in mind while they write the code, they can use them to reduce the effort they need to put into writing automated tests afterwards [15].

In the industry the ISO/IEC 9126 [23] standard is used to assess whether software has reached a certain degree of quality. For example the German TÜV (an association for technical inspection, similar to the MOT) uses this standard to rate software [24]. However, the rules defined in this standard cannot be measured directly. In [25] Correia et. al propose a mapping of software metrics to the categories defined by the ISO/IEC 9126 standard. They evaluate their mapping using a survey based approach, for which they questioned a group of software experts. As a result, they identified a set of measurements which can be directly linked to the aspects defined in the standard. In this thesis we also took these results into consideration while we selected the metrics we are going to use.

During the preparation of this thesis we also evaluated how certain metrics correlate with each other, as also shown in [26]. We tried to minimise redundancy in our measurements as much as possible. Besides that, we chose metrics, which are considered the most relevant when designing software according to [5, 9, 10, 21]. As a result of this selection we sorted out a large set of metrics that were invented to be applicable to only certain languages and to examine specifics of those languages. The framework presented here is intended to be applicable to a larger set of languages without major difficulties as it uses metrics that are to a great extent not bound to specific programming languages.

2.2 McCabe Complexity

The McCabe complexity or cyclomatic complexity measure was introduced in 1976 by Thomas J. McCabe during a time when programs were mostly measured based on their physical size [6]. McCabe argued that even a relatively small program with respect to lines of code can easily become hard to maintain or understand. Moreover he claimed that the complexity of code is strongly influenced by how many control flows exist and not by the volume of the code. Given a program consisting of fifty lines of code and that contains twenty-five consecutive if-then statements, then this program can have 2^{25} ($\approx$ 33.5 million) distinct control paths. McCabe therefore concentrated on measuring the amount of control paths inside a program using principles of graph theory. The metric operates on an abstract syntax tree and therefore does not rely on a specific programming language [7]. This results in a wide tool support of this metric for most popular programming languages. The cyclomatic complexity v of a program G is defined

as:

$$v(G) = e - n + 2p$$

where e, n, and p are:

$$e = \text{number of edges in the graph,}$$
$$n = \text{number of nodes in the graph, and}$$
$$p = \text{number of connected components (exit nodes)}$$

Figure 2.1 shows basic programming principles like if-then-else or while and their corresponding cyclomatic complexity values. For example the graph for an if-then-else statement as shown in Figure 2.1b has $e = 4$ edges that represent the split introduced by the if-statement and the following merge, $n = 4$ nodes, and $p = 1$ exit nodes. Therefore the cyclomatic complexity $v(G)$ is computed as $v(G) = 4-4+2 = 2$. The value $v(G) = 2$ tells us that there are two linearly independent paths in the graph for the application. Furthermore this complexity measure represents the minimum number of test cases that have to be written in order to test the of code. This implies that each test case will always test exactly one possible path through the application.

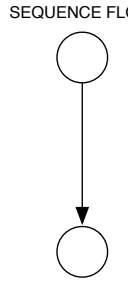
The cyclomatic complexity metric $v(G)$ has the following properties:

1. $v(G) \geq 1$, because even a function that immediately returns has exactly one edge, two nodes, and one exit node as shown in Figure 2.1a.
2. $v(G)$ is the number of linearly independent paths in G.
3. Inserting or deleting functional statements into G does not affect $v(G)$.
4. G has only one path if and only if $v(G) = 1$.
5. $v(G)$ only depends on the decision structure of G.

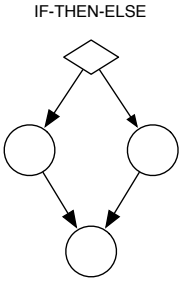
We compute the McCabe complexity on the function level and aggregate the values to draw conclusions on a class or file level.

2.3 Halstead Complexity

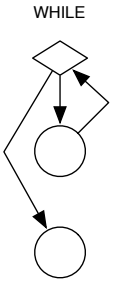
In 1977 Maurice Howard Halstead introduced a set of metrics which are now known as the Halstead metrics [8]. Contrary to the cyclomatic complexity introduced by Thomas J. McCabe (see Section 2.2) Halstead did not focus on the complexity introduced to computer programs by adding conditions to the control flow. His assumption was that a



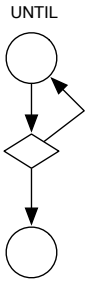
(A) $v(G) = 1 - 2 + 2 = 1$



(B) $v(G) = 4 - 4 + 2 = 2$



(C) $v(G) = 3 - 3 + 2 = 2$



(D) $v(G) = 3 - 3 + 2 = 2$

FIGURE 2.1: Flow graphs for different control flow structures with their respective complexity values.

computer program mainly consisted of two parts: operators and operands. The complexity will increase whenever new operators or operands are introduced. It will always be harder for a developer to understand or maintain code with a lot of different operations or variables. While programming he or she must keep the complete set of variables in mind in order to use them appropriately. In summary, Halstead based his metrics on these four key variables.

$$\begin{aligned} \eta_1 &= \text{number of distinct operators,} \\ \eta_2 &= \text{number of distinct operands,} \\ N_1 &= \text{total number of operators, and} \\ N_2 &= \text{total number of operands} \end{aligned}$$

One can see, that in order to apply this measure to a given piece of code a precise definition of what is an operator and what is and operand in a given language has to be present. Given the variables η_1, η_2, N_1, and N_2 Halstead derived a set of metrics.

- Programming vocabulary: $\eta = \eta_1 + \eta_2$
- Program length: $N = N_1 + N_2$
- Volume: $V = N \times \log_2 \eta$
- Difficulty: $D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}$

- Effort: $E = D \times V$

In this thesis we focus on the metrics "difficulty" and "volume". This decision is based on research such as [27]. As there has been much discussion around the effort metric [16] and whether it can truly be used to make assumptions, we chose to not use it at all in our examinations. Despite these discussions it has also been proven that the Halstead metrics can be used in order to predict the maintainability of software [16].

For completeness we want to state that we are aware of other measures such as "Implementation Time" $T = \frac{E}{18}$ in the suite of metrics which are based on the Halstead metrics, but chose to not include them for the same reasons which applied when ruling out the effort metric.

We now want to give a brief example of how the Halstead values are computed using the code shown in Listing 2.1.

```
1  function sort(list) {
2      if(list.length < 2) {
3          return;
4      }
5
6      var i, j, tmp;
7
8      for(i = 0; i < list.length - 1; i = i + 1) {
9          for(j = i + 1; j < list.length; j = j + 1) {
10             if(list[i] > list[j]) {
11                 tmp = list[i];
12                 list[i] = list[j];
13                 list[j] = tmp;
14             }
15         }
16     }
17 }
```

Listing 2.1: JavaScript sort function as an example for the computation of Halstead's metrics.

As presented in Tables 2.1 and 2.2 there is a total of 60 operator and 35 operand occurrences in the source code of Listing 2.1. These are the values for N_1 and N_2 respectively. One might wonder why almost every single character is listed on its own except for "[ ]". In order to measure Halstead's metrics on a given language it is necessary to define beforehand what an operator is and what not. For the example at hand we rely on a definition given in [28]. We argue that contrary to symbols like "{" and "}" or "(" and ")" which surround blocks of code, the access to an array-like structure using "[" and "]" is always restricted to a very small piece of code. Therefore "[" and "]" are listed as one

Operand	#	Operand	#	Operand	#	Operand	#	Operand	#	Operand	#
return	1	–	1	,	2	<	3	)	4	[ ]	6
var	1	if	1	length	3	+	3	{	4	=	7
>	1	for	2	.	3	(	4	}	4	;	9

Table 2.1: List of operands inside the source code of Listing 2.1

Operator	#	Operator	#	Operator	#	Operator	#
0	1	tmp	3	j	8	list	9
2	1	1	4	i	9		

Table 2.2: List of operators inside the source code of Listing 2.1

operator. This does not apply to every possible programming language. The values for η_1 and η_2 are 18 and 7. Using these values we can compute the metrics as follows.

$$\begin{aligned} \eta &= \eta_1 + \eta_2 && = 25 \\ N &= N_1 + N_2 && = 95 \\ V &= N \times \log_2 \eta && \approx 441 \\ D &= \frac{\eta_1}{2} \times \frac{N_2}{\eta_2} && = 45 \\ E &= D \times V && \approx 19845 \end{aligned}$$

In our analysis we compute the Haltstead metrics on the function level as we do it with the McCabe complexity shown in Section 2.2. We then aggregate the individual values in order to draw conclusions on the class or file level.

2.4 Coupling

Metrics that deal with coupling reason about how software is structured. Literature agrees that software is easier to understand and to maintain if it is split into modules of reasonable size [12]. We therefore decided to include metrics that measure the structure of software into our measurement framework.

However, in order to measure the structure and the interaction that happens inside software, a programming language that embraces these concepts is required. Languages as Python[1] or Java[2] are designed in a way that packages and modules form an integral part of the respective language. If you write code inside one module and want to use

[1] `http://www.python.org/`
[2] `http://www.java.com`

functionality from another one you have to explicitly import these modules. This way it is possible for a tool which measures the structure of the software to identify how modules interact with each other. On the other hand, languages like JavaScript[3] do not enforce such clear definitions. This is mainly based on the fact that JavaScript has been developed primarily to be used in web pages during a time when people did not expect complex web applications, as we see them today, to be feasible [29]. In the future we might see a change in this as JavaScript more and more finds its way into backend development as well and frameworks as requireJS[4] mimic imports as we know them from Java or Python.

Even though coupling can also be measured on a per function basis, we decided to measure this metric on class level, as this can be regarded as the sum of all functions.

There are two types of coupling.

- Efferent coupling C_e, also known as Fan-Out and
- Afferent coupling C_a, also known as Fan-In.

Due to their naming it is likely to confuse one with the other. But there is an easy way to remember the meaning of each metric. Efferent coupling C_e describes how much a given class C depends on other classes. Therefore C will receive the effects of the changes made in the other classes. A high value in this metric is typical for classes that do orchestration. Thus a high value does not necessarily represent bad design but simply the nature of things. Nevertheless it might also be an indicator that said class C has too many responsibilities. In this case C_e should be reduced by splitting C into several smaller classes where each class handles one aspect of C.

Afferent coupling on the other hand expresses how many classes depend on a given class C. Therefore changes in C will affect all classes which depend on it. This in turn can also be an indicator of good design and code reuse.

High values in either case can therefore never be seen as iron-clad but merely as an indicator. But classes which have high values for both C_e and C_a are often a source of bugs [18].

Besides C_a and C_e, there also exist a wide range of object oriented metrics as described in [20]. But as stated in Section 1.1 we tried to keep the metrics we choose to be as generally applicable as possible. For example the Method Hiding Factor (MHF), or the Attribute Hiding Factor (AHF) would impose even more restrictions on the set of

[3] Also known as ECMAScript `http://www.ecmascript.org/`
[4] `http://requirejs.org/`

languages that can be measured. In order to compute the MHF or AHF classes inside a language need to be identifiable throughout the whole software system, which is a hard job if the language is not statically typed. Also the concept of public and private properties has to be incorporated, because otherwise all methods and attributes would be visible to all classes. This in turn would render these metrics useless.

Chapter 3

The Analyzr Framework

To the point of this writing there existed no software known to us which was capable of performing the tasks required in order to gain the needed insights into source code management (SCM) systems. Therefore it was necessary to develop a framework[1] which could perform all steps needed to take the measures on the source code. As different organisations use different SCM systems, the framework must abstract from these systems in order for the measurements to work on a unified view of a repository. Furthermore for every system it must be possible to retrieve information concerning the number and location of different branches, authors who contributed to a repository, all versions created in the different branches, and for each revision which files were modified. After this basic data has been gathered further measurements have to take place. As it was neither possible nor desirable to implement all measurements individually the framework makes use of third party tools which are described in more detail in Section 3.4. The data produced by these tools is extracted from the SCM system, transformed into our data model, and saved so that analysis can be performed on a well defined data structure (see Section 3.1). After all data has been collected and the measurements have taken place, the framework is able to produce a graphical representation of the data by providing a web interface which can be used to explore each repository.

The framework is composed of two major parts. The backend is written using Django[2]. It abstracts from the different version control systems and code metric tools and collects all the data. The frontend is a so-called HTML5 application powered by JavaScript that can be accessed via the browser. This kind of architecture has been chosen as analysis and measurement of a repository can be a time consuming task. If we would have implemented the framework as a standard Desktop application the computer that runs the analysis would have to be online as long as the measurements are taken. Having a

[1] `https://github.com/pxlplnt/analyzr/`
[2] `https://www.djangoproject.com/`

web application running on a remote server makes it easy to run any task asynchronously and being able to access the data regardless of ones current location.

3.1 Data Model

In the past a lot of preprocessing had to be done in order to being able to process data retrieved from SCM systems [30]. There were no bindings available for programming languages that could be used to access the information inside an SCM system in a structured way. One needed to parse every log entry of an SCM system and then manually extract the data that should be analysed. Obviously this method is prone to error.

Luckily this situation has changed in the recent years so that we could profit from advanced libraries that make it easy to access all data inside a given SCM system. However, we still needed to extract all required data and transform it into a consistent data model as shown in Figure 3.1. The data model reflects the basic structure of a software repository. On the top there is the repository entity which holds information such as the kind of the SCM system, the location of the remote repository, and the user credentials. A repository can have an arbitrary number of branches. In each of these branches authors can create revisions that consist of a set of changed files. For the sake of query performance we decided to keep some information, as for instance the author, redundantly in both the file and revision tables. The tables for revision and file data have by far the largest cardinality. If we would have decided to make the author of a file only accessible through the revision to which the file belongs to, it would mean, that we have to join both tables in order to get a result. Especially for repositories with a large number of revisions this would have had a drastic effect on query performance. Additionally, information regarding packages (i.e. folders) is stored. The package information is used in order to aggregate metric values for a set of files. Otherwise it would only be possible to view the metric values on a per-file basis or for the whole repository. As packages are usually represented in a tree-like structure which can dramatically reduce the query performance, we now want to elaborate more on how they are represented in our framework to overcome this issue.

3.1.1 Package Structure

In most cases software is structured into modules. Those modules are usually represented by folders in the file system. We wanted an efficient way to aggregate information of all files of certain modules. Thus we needed a way to aggregate all files not only within one folder, but also all files which are contained in all subfolders. The problem with

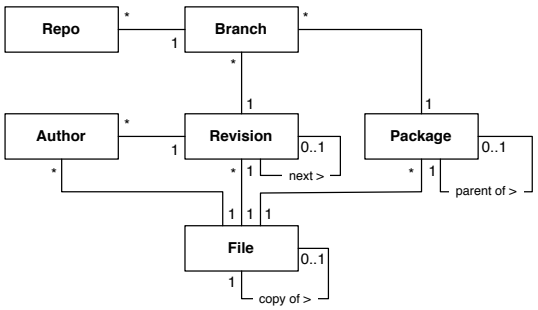


FIGURE 3.1: The data model used to store gathered information related to repositories.

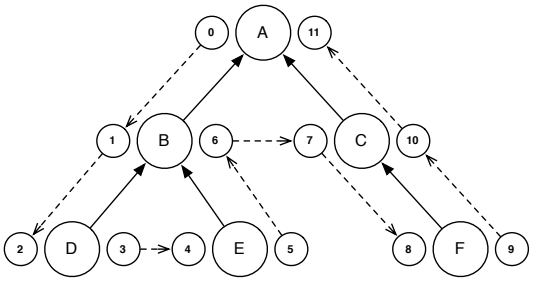


FIGURE 3.2: Data model for packages which incorporates aspects of the naïve tree implementation and nested sets to speed up read operations.

most models that are used in order to represent tree structures in relational databases is that they are either read-optimised or write-optimised. We needed an approach that is fast for read as well as write operations under the given circumstances. Therefore we leveraged the knowledge we had about the data we are dealing with and created a hybrid solution depicted in Figure 3.2.

Once we have scanned a repository the data will not change anymore. We can therefore use a naïve implementation where every child node holds a pointer to its parent node while we are inserting nodes to the tree. Would we use the same method to read the whole tree we would run into performance issues as a multitude of joins would be necessary in order to retrieve all descendants of a given node. After the initial analysis of a repository is done and all packages have been saved to the database we build a nested set representation [31] of the tree. This means each node has two additional properties "left" and "right". As shown in Figure 3.2 we then start numbering at the root node and follow child relations. Doing this yields the advantage of being able to very easily query subsets of a tree. For example all children of node B can be found by querying for nodes where left > 1 and right < 6. We can also easily decide if a given node is a leaf of the tree. In this case right $=$ left $+ 1$ is true.

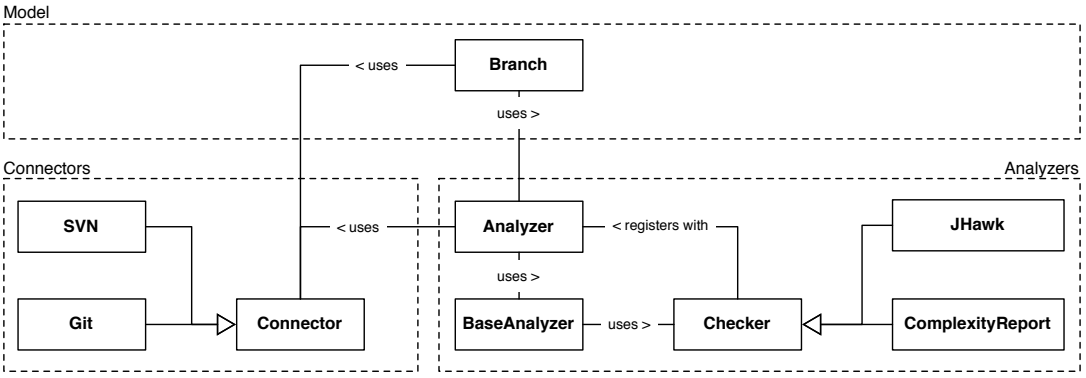


FIGURE 3.3: Backend architecture which shows how the three parts of the backend interact with each other.

3.2 Architecture

As we intended the analyzr framework to be extensible and wanted to give people the opportunity to present data the way they like to, we decided to split the framework into two separate components. The backend concerns itself with data collection and exposes a REST[3] interface through which the data is accessible. The frontend uses the REST interface and presents a user interface to explore the data.

3.2.1 Backend

The backend connects our application to the different repositories that are analysed and holds the data which is gathered during our analysis. It is written using Django and follows the model-view-controller (MVC) design pattern [32]. We followed the principle of "fat model, skinny controller" in order to keep the logic as close as possible to the data it operates on. Figure 3.3 therefore does not depict the different controllers or models. Every model is represented by a controller in order to access the respective resource. In order to carry out our analysis the branch model assumes a special role. Analysing a repository (i.e. gathering general information about revisions and the structure of a repository) and measuring the metrics afterwards is always triggered through a branch instance.

An abstract connector class is used in order to communicate with the SCM system which is associated to the repository of the given branch. We currently are able to connect to Subversion[4] and Git[5] repositories. Every specialised connector is then able to extract all relevant information from each revision of the given branch. As the branch instance

[3] `http://en.wikipedia.org/wiki/Representational_state_transfer`
[4] `http://subversion.apache.org/`
[5] `http://git-scm.com/`

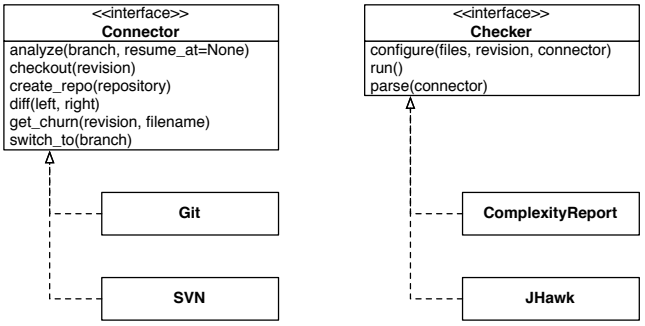


FIGURE 3.4: The interfaces which have to be implemented when adding new connectors or checkers to the analyzr framework.

only interacts with the abstract connector class, new SCM systems can be easily added by implementing a small interface as shown in Figure 3.4.

After this step is completed the metrics can be measured using the collected data. The analyzer class will go through all revisions that are present for a given branch, check out the files using a connector instance, and then use the specific checkers to perform the measurements. To measure the metrics, we rely on third party software (c.f. Section 3.4). The checker instances wrap the third party software so that they expose a common interface (see Figure 3.4). This allows us to easily add new functionality to the framework without changing the general logic. Also multiple checkers are allowed for one language. This way even if one single third party library cannot measure all metrics it is still possible to achieve this overall goal by incorporating multiple measurement tools for one language. The checkers themselves register with the analyzer class, so again the general logic does not need to be modified when a new language is added. Also with each checker the first step of the analysis where the data is read from the appropriate SCM system will be extended. To not clutter the database with useless information we only save information for files that can be measured afterwards. This information in turn is gathered by looking at the checkers that are registered in the system.

3.2.2 Frontend

The frontend is implemented as an HTML5 web application and utilised to present the results of our analysis and to manage the repositories we want to examine. We are able to track the progress for all repositories that are currently registered with our framework as shown in Figure 3.5. The communication with the backend happens via a REST interface.

The frontend itself is composed of several different components as shown in Figure 3.6. Each component is linked to a certain resource on the backend side. It will then query

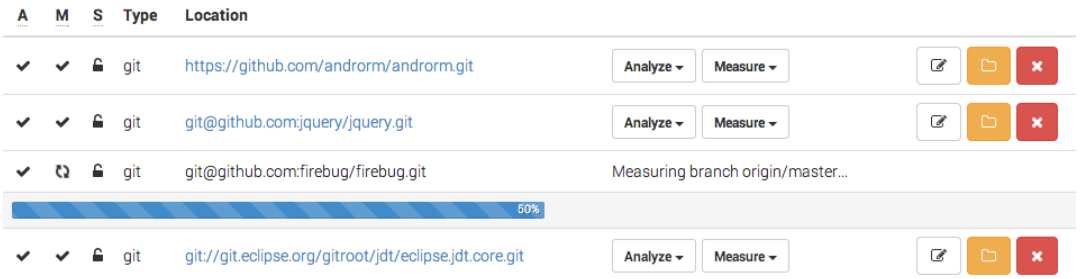


FIGURE 3.5: An example overview of repositories which can be accessed using the Analyzr framework.

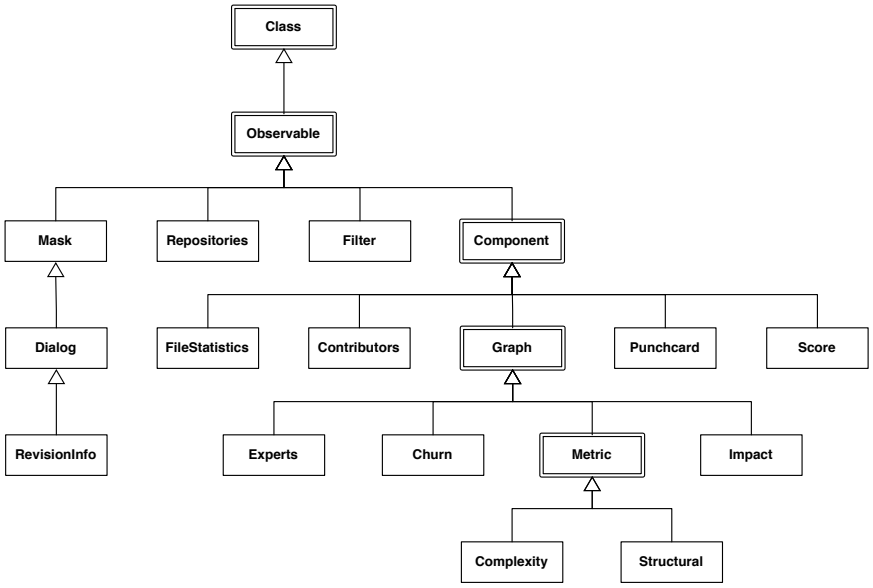


FIGURE 3.6: Inheritance tree of the components used in the frontend.

the appropriate interface for data and display the data in the intended way. As not only the format to access resources but also the format for result data for queries to the backend is clearly defined, most components can transparently handle queries which are specific for a branch or a specific author. For example the graph for complexity metrics will query and display the data for a whole branch if only the branch is supplied. On the other hand if a branch and an author are specified it will modify the request and display data specific for the given author on the given branch. This design makes it feasible to easily exchange and add components to different pages in the frontend.

In Figure 3.6 we represent classes which are meant to be abstract with a double outline. For example the observable class exposes an interface for event handling which is used by many components in order to communicate with each other. The component class handles most of the server communication for all classes which inherit from it. It is able to determine whether the scope of a component is solely the branch or if information

regarding a specific author is required. We use the d3[6] framework when we present data in the form of graphs. As there is a lot of setup work to be done, which is the same for every graph we need to draw, the graph class abstracts from that, which made it possible to develop different kinds of visualisations without much overhead.

3.3 Aggregation of Code Metrics

We decided to store metrics on a per file basis. This decision is on the one hand based on the idea that package-level metrics are too generic and we probably could not observe changes for single developers. On the other hand we decided against storing metric information on a per-method-level because we would then run into the problem of tracking and linking changes in method signatures. For example if a developer renames a method we would have to determine that this is not a new method and the old one has been deleted but that nothing has changed at all. Still, understanding these processes might be able to enhance our work in the future.

However, the tools we use to measure the source code produce metric values on a per function level. So in order to store the results we needed to aggregate these low-level marks into one value. The source lines of code (SLOC) can be easily aggregated using the sum of all values [33]. In an early version we additionally used mean and also tried to improve the values using median but found that those values correlate much as also shown in [34]. We found that regular aggregation methods are not suitable for code metrics. Most code metrics have their own domain and therefore are hard to compare. To overcome these issues we decided to use the Squale [35] model to aggregate metric values and want to describe it in more detail in the following section.

3.3.1 Software Quality Enhancement (Squale)

The squale model for aggregating software metrics as described in [35] overcomes issues that exist when using classical aggregation mechanisms for code quality metrics. It highlights problematic values and also provides stronger feedback if those values are corrected. Also it shows improvements of small parts in a generally all-bad system and provides a bounded, continuous scale that makes it easier to compare metric values.

Squale consists of two parts: Low-level marks and high-level marks.

Definition 3.1 (Low-level mark)**.** Low-level marks are the raw pieces of information which can be retrieved from the source code. These are either manual metrics which are

[6] `http://d3js.org/`

Metric	Computation	Metric	Computation
Cyclomatic Complexity	$\text{IM}_{cc} = 2^{(7-cc)/3.5}$	Source Lines of Code	$\text{IM}_{sloc} = 2^{(70-sloc)/21}$
Halstead Volume	$\text{IM}_{hv} = 3 - \frac{3 \times hv}{1000}$	Afferent Coupling	$\text{IM}_{C_a} = 2^{(30-C_a)/7}$
Halstead Difficulty	$\text{IM}_{hd} = 3 - \frac{3 \times hd}{50}$	Efferent Coupling	$\text{IM}_{C_e} = 2^{(10-C_e)/2}$

Table 3.1: Overview how individual marks are computed for certain metrics.

assessed by humans or raw metrics that are measured using code metrics, rule checking, etc.

Definition 3.2 (High-level mark)**.** High-level marks are computed from low-level marks. Good or bad values with respect to project quality are determined by experts for the given field. The values which are produced all lie in the same domain in order to be comparable.

After each high-level mark or individual mark (IM) has been computed they can be aggregated. Before that happens the weighting function g is applied. A hard weighting gives more weight to bad results than a soft weighting. The function is defined as:

$$g(\text{IM}) = \lambda^{-\text{IM}}$$

λ defines the strength of the weighting. Common values are 3, 9, and 13 for soft, medium, and hard respectively.

The global mark GM^{λ} is computed as follows:

$$\text{GM}^{\lambda} = -\log_{\lambda}\left(\frac{1}{n} \cdot \sum_{i=1}^{n} g(\text{IM}_i)\right)$$

where n is the total number of individual marks.

To compute the individual marks for the metrics presented in Chapter 2 we used the following computations, which are described in [36]. An overview of all methods can be found in Table 3.1.

Cyclomatic Complexity We consider code with a complexity value less or equal to 2 to be not complex at all. This includes methods with no branches and the ones with for example only a simple type check in them. Methods with a cyclomatic complexity equal to or greater than 20 are considered very complex. The individual mark IM_{cc} is

computed as:

$$\mathrm{IM}_{cc} = \begin{cases} 3 & \text{if } cc \leq 2 \\ 0 & \text{if } cc \geq 19 \\ 2^{(7-cc)/3.5} & \text{otherwise} \end{cases}$$

Halstead Volume & Halstead Difficulty As [36] did not consider the Halstead metrics we assumed an even mapping from the metric values to the values for the individual marks. Based on [28] we define every value for the volume metric which is less or equal to 20 to be not complex and every value greater or equal to 1000 to be very complex. For the difficulty metric the values are 10 and 50 respectively. The individual marks IM_{hv} for the Halstead Volume or IM_{hd} for the Halstead Difficulty are computed as:

$$\mathrm{IM}_{hv} = \begin{cases} 3 & \text{if } hv \leq 20 \\ 0 & \text{if } hv \geq 1000 \\ 3 - \frac{3 \times hv}{1000} & \text{otherwise} \end{cases} \quad \text{and} \quad \mathrm{IM}_{hd} = \begin{cases} 3 & \text{if } hd \leq 10 \\ 0 & \text{if } hd \geq 50 \\ 3 - \frac{3 \times hd}{50} & \text{otherwise} \end{cases}$$

Source Lines of Code Source lines of code are not only tracked in order to measure the impact of a developer but are also an indicator for code quality. Based on Halstead's metrics larger methods are harder to understand than smaller ones. Therefore we also compute marks for the length of methods. Once again we comply with the work of [36] when we define the ranges. Every method with a length less or equal to 37 is considered not complex and methods with 162 or more lines of code are considered very complex. The individual mark IM_{sloc} is computed as:

$$\mathrm{IM}_{sloc} = \begin{cases} 3 & \text{if } sloc \leq 37 \\ 0 & \text{if } sloc \geq 162 \\ 2^{(70-sloc)/21} & \text{otherwise} \end{cases}$$

Afferent Coupling To recall: Afferent coupling C_a describes how many classes depend on a given class C. Even though it is good practice to reuse code, a class becomes harder to maintain the more classes depend on it. According to [36] a value of up to 19 is considered very good. Everything greater than 60 should be thoroughly reviewed. The individual mark IM_{C_a} is computed as:

$$\mathrm{IM}_{C_a} = \begin{cases} 3 & \text{if } C_a \leq 19 \\ 0 & \text{if } C_a \geq 60 \\ 2^{(30-C_a)/7} & \text{otherwise} \end{cases}$$

Efferent Coupling Complementary to the afferent coupling, the efferent coupling C_e describes how much a given class C depends on other classes. Classes which depend on too many other classes are likely to need more maintenance because they receive the effects of the changes which were made in the classes they depend on. Here once again following [36] a value of up to 6 is considered as very good and everything above 19 should be reviewed. The individual mark IM_{C_e} is computed as:

$$\mathrm{IM}_{C_e} = \begin{cases} 3 & \text{if } C_e \leq 6 \\ 0 & \text{if } C_e \geq 19 \\ 2^{(10-C_e)/2} & \text{otherwise} \end{cases}$$

During the analysis the global mark GM will be computed using the individual marks of each metric. However, we do not aggregate the individual marks of all metrics into one general mark but store the general mark for each kind of metric.

3.4 Third Party Tools

After we have presented the framework we built in order to manage all repositories which should be analysed, we want to present the third-party tools we use to perform the actual measurements on the source code. It would have neither been possible nor sensible to create a framework completely from scratch which is able to perform those tasks in the scope of this thesis. We therefore incorporated different third party tools to measure the metrics. This section gives an overview of the tools we used and also highlights why we have chosen each tool over the possible competitors.

3.4.1 JHawk

A lot of tools are available on the market that measure metrics on java source code. They differ mostly in regard to their focus and support for different metrics. With a statically typed language such as Java it is possible to measure many metrics ranging from generic complexity metrics to object oriented metrics as described in [20]. Unfortunately, this also reflects on the performance of each tool. As we knew that we had to operate on a multitude of revisions for each repository, the tool had to be both fast and equipped with the capabilities to measure all the metrics which are relevant to us. The advantage we have is that all of our required metrics can be calculated using only the bare source code. This means the code does not need to be compiled before we can run the analysis. We could therefore speed up the whole process as we did not need to worry about certain

dependencies inside the source code of the repositories at hand that might prevent the project to build and therefore would render any analysis unfeasible.

Programs like checkstyle[7] or PMD[8] are popular tools, which can be used to measure a set of code metrics on Java source code. However, both tools are not able to measure all the metrics we require. Checkstyle can be used to measure the cyclomatic complexity of source code and the efferent coupling C_e of a class, but lacks the support for the required Halstead metrics and afferent coupling C_a. PMD is able to measure the cyclomatic complexity but defines metrics regarding the coupling between classes or the code size only in a way that makes it not possible for us to include them into our framework. Therefore both tools have been ruled out during our evaluation.

With JHawk[9] we found a tool which offers support for a wide range of software metrics[10]. It also does not require to build the whole project before running its measurements. This increases the speed of the analysis a lot. JHawk can be run using ant[11], invoking the ant task which is provided by JHawk. It is possible to start measurements in a given directory and restrict the set of files which will be analysed. Using this mechanism we could incorporate the knowledge we have about changed files using the collected revision data. We exactly know which files have been modified in a given revision and therefore can perform the calculations on these files and do not have to measure the whole codebase. JHawk then produces an XML document that we further processed to feed the gathered data back into our system. The downside of the huge amount of metrics which are computed by JHawk is that the file size of the generated XML report can quickly get out of hand. Especially with merge revisions we found that the size of the XML documents exceeded the main memory size of our test servers. Therefore we implemented a limit which ensures that JHawk will only process a certain amount of files at a time.

We have to note that JHawk is a commercial product and can therefore not be bundled with our framework.

3.4.2 Complexity Report

In 2002 Douglas Crockford first released JSLint, a tool to identify code smells inside JavaScript and help developers to write better code. Unfortunately JSLint is strongly biased towards Crockford's opinion on how to write good code [29]. In the following

[7] `http://checkstyle.sourceforge.net/`
[8] `http://pmd.sourceforge.net/`
[9] `http://www.virtualmachinery.com/`
[10] `http://www.virtualmachinery.com/Jhawkmetricslist.htm`
[11] `http://ant.apache.org/`

years a project called JSHint was started forking away from the original JSLint source and creating a more tolerant and better configurable version of JSLint. We encourage everybody working with JavaScript to have one of these linters check their code during development.

However, both tools are not able to measure the required Halstead metrics which is the reason we chose Complexity Report[12] to asses JavaScript. It is an open source software that is available as a node.js[13] package and that can be run from the command line. As JavaScript is a loosely typed language which does not natively facilitate the concept of packages we are not able to measure C_e and C_a. Nevertheless we are able to measure the McCabe complexity, the Halstead metrics, and the SLOC using Complexity Report.

In theory this tool is able to measure a set of files at a time. Unfortunately, if a syntax error exists in only one of the files the whole process fails. We therefore use it in a way where we measure each file separately which makes it less prone to errors.

In summary we are able to measure Java and JavaScript projects utilising the largest set of our required metrics that is applicable to the respective language.

[12] `https://github.com/philbooth/complexity-report`
[13] `http://nodejs.org/`

Chapter 4

Use Cases

In the last chapters we have shown the importance of code metrics and which ones we found most relevant for a general analysis. We also introduced the framework we are going to use to analyse code repositories. Unfortunately, we would not be able to validate our results, if we would measure code repositories where we do not have any insights into the structure of the organisation.

This chapter will introduce the different software projects and organisations we will examine during our analysis. It should help to put the analysis in perspective and also aims at explaining why we have decided to focus our analysis on the presented projects. Table 4.1 summarises the relevant information for all repositories.

4.1 Signavio GmbH

The Signavio GmbH[1] was founded in May 2009 as a spin-off from the ORYX project [37] of the Hasso-Plattner-Institute. We chose Signavio because the author is working there since August 2009 till the present day and therefore has deep insights into the structures and processes within the company. Approximately thirty of its fifty employees are actively developing the software. Of these thirty around ten can be considered frontend developers and the other twenty are concerned with backend development. It is therefore possible for us to validate the assumptions we make in this thesis using Signavio as a case study. Also errors in the algorithms and data structures of the framework we use could be found and removed in a very fast manner thanks to the immediate feedback we got at Signavio.

[1] `http://www.signavio.com`

Signavio offers a web-based Business Process Management tool which enables corporations to collaboratively work on its processes. The backend of the software is implemented in Java and the frontend is a JavaScript web-application that is decomposed into several applications. This makes it possible for us to determine experts on a per-application level.

In compliance with German data privacy laws all names of the developers have been changed. This needs to be emphasised as the names we are going to use will still be regular human names. We chose to do so for the sake of readability. Nevertheless any resemblance to real persons is absolutely coincidental and not intended by the authors.

As Signavio grew without venture investments the software needed to reach a certain amount of functionality very fast so that it would be attractive to customers. We therefore would expect lower code quality in terms of software metrics especially in the beginning. New features needed to be added and there was only very little time to refactor the code.

In the last two years the company focussed more on creating code which can be better maintained as a lot of new developers are constantly joining the team and the effort to train them should be minimised. Also, as the software grew it became more and more important to get a higher test coverage both for the front- and the backend in order to keep the product viable while still maintaining a high degree of innovation.

Currently measuring code metrics is not part of the development process at Signavio which is organised in a Scrum-like [38] approach. Sprints take three weeks and are split into an implementation and a testing phase of equal size. In the testing phase the developers write functional tests for their own tickets and perform user tests for tickets of other developers. In addition to those tests code reviews are performed amongst developers. This also aims at keeping the code maintainable as difficult parts are likely to be spotted before the code goes into production.

To track changes in the codebase Signavio uses SVN as their SCM system.

4.2 jQuery

JQuery[2] is a cross-platform JavaScript library designed to simplify client-side scripting of HTML. It was initially released in 2006 by John Resig and is currently developed under the supervision of Dave Methvin. The library is used by 80% of the 10,000 most popular web pages which makes it the most popular JavaScript library in use [39].

[2] `http://www.jquery.com`

Project	First Commit	Age	# Revisions	# Authors
Signavio	April 9, 2009	4 years, 262 days	14336	42
jQuery	March 22, 2006	7 years, 328 days	4798	228
Eclipse	June 5, 2001	12 years, 254 days	19344	84

Table 4.1: Overview of the repositories which serve as a use case for our analysis as at February 18, 2014

We have chosen jQuery based on several reasons. As being so popular and widely used jQuery also must be very reliable and robust. In the world of web development this means dealing with a lot of different browser quirks. jQuery abstracts from the differences that exist between browsers and exposes a simple interface. Also on April 18, 2013 the jQuery team released a version which dropped all support for Internet Explorer versions 6 through 8. One would assume that dropping support for three entire versions of a browser would decrease the complexity in the code drastically as a lot of code which was concerned only with managing the quirks in those version should have been removed from the project.

The jQuery project is currently hosted on GitHub[3] and at the point of this writing a total of 228 authors had contributed code to the project resulting in 4798 revisions.

4.3 Eclipse JDT

For the third project we chose the Java Development Tools (JDT) of the Eclipse project. Eclipse is an Integrated Development Environment (IDE) initially developed by IBM under the name IBM VisualAge[4]. The software is now being developed for over ten years by over eighty developers. Eclipse is written in Java and built in a way so that it is customisable by installing different plugins.

We chose this project because we think it is very interesting as it combines two worlds. The project started as proprietary software of IBM and was the open sourced. This way two different cultures influenced the software and also the making of it. A very interesting aspect would be to see if and how the metric values changed when the software stopped being proprietarily developed by IBM and became an open source project. We would expect a decrease of the quality metrics as new developers that do not work full-time on the project and do not have deep insights in the project structure have started contributing code. On the other hand a review process could have been established to tackle this problem which would probably result in a constant quality of the code.

[3] `https://github.com/jquery/jquery`
[4] `http://wiki.eclipse.org/FAQ_Where_did_Eclipse_come_from`

Chapter 5

Related Work

This thesis is in great parts inspired by the research that has already been done. In this chapter we want to highlight some of the key publications which influenced our decision making process. Related work will be presented separated into three topics which cover the main challenges we had to tackle while working on this thesis. First we will present work on the topic of how software metrics can be applied and which common pitfalls should be avoided. After that we will present work regarding the aggregation of software metrics. This topic is essential to this thesis and is also discussed in more detail in Section 3.3. The last section of this chapter presents related articles about finding experts inside software projects. This topic is especially interesting as not only do we need to cover solely technical aspects but also have to have a look from a sociological perspective. How developers work and what aspects influence their day-to-day business will be of interest in this section.

5.1 Measuring Source Code

Coleman et. al [16] evaluate different software metrics in order to find a means to predict the maintainability of software systems. In order to do so they use the HP software maintainability assessment system (HPMAS) which is a hierarchical model that divides maintainability into three dimensions.

Control Structure which describes how the program is decomposed into algorithms,

Information Structure includes characteristics describing the choice of data structures and data flow techniques, and

Typography, naming, and commenting which includes characteristics concerning the general layout of the code.

In order to measure metrics that fall into the categories "Control Structure" and "Typography, naming, and commenting" the authors use the McCabe complexity (c.f. Section 2.2) and a set of Halstead's metrics (c.f. Section 2.3) respectively. It is not stated how measurements regarding the "Information Structure" are taken but we argue that metrics as described in Section 2.4 of this thesis could be used.

Furthermore it is shown, that the Halstead metrics provide a good means in order to assess maintainability of a software system. The results of their analysis are evaluated by interviewing real world experts for the systems which have been analysed with a focus on maintainability. It is stated that in most cases the analysis captured the *intuition* of the respective experts with regards to the perceived maintainability of a system or component. Also the authors clearly state that this approach should be used in order to "[...] help maintainers guide their efforts and provide [the developers] with much needed feedback."

This model yields insights into the current state of a software system, yet it is not able to predict which parts of a software system are becoming increasingly complex and thus harder to maintain. Even worse, actions might only be taken when the "felt" complexity of a component reaches a certain point. This way developers can only react when its too late and the system has already become very complex. In our approach actions could be taken beforehand so that components with an increasing trend in complexity could be refactored before the complexity gets out of hand.

Clark et. al [19] use software metrics in the field of autonomous vehicles. As one can imagine a software system needed to operate a car can easily get very complicated. At the same time it must be easy enough to maintain so that as few bugs as possible find their way into the software. The authors state that measuring software attributes can either be (a) measuring the size and content or (b) the complexity of the software. Even though the raw size of code measured as lines of code does give a fast measure for the size of the code base one can neither tell how difficult to understand the code is nor what the algorithmic complexity of the code is. Concerning the set of metrics they rely solely on the McCabe complexity (c.f. Section 2.2) as case studies have shown a high correlation between code errors and a high McCabe complexity [7].

With our approach of observing metric values over time we can derive a trend for certain parts of software and therefore see where actions are underway to reduce complexity and where parts of the software are going to become increasingly complex. Even though this approach will give insights into the current state of a software project, it cannot predict which parts might become problematic in the future. We also argue that relying only on the McCabe complexity neglects the complexity which is added by very large methods.

If a method does not fit into the window of a regular editor it is not possible for the developer to grasp it as a whole. This is likely to introduce bugs as the developer cannot oversee how variables change when the method is executed.

Nagappan et. al [17, 18] correlate code metrics and code churn with the probability of defects in a given software system. The authors state that there exists a "reliability chasm" of the observed quality of software in its pre-release state and its post-release use in the field. In order to find parts inside the software that are prone to produce defects they use two approaches.

The first approach focusses on software metrics [18]. They state that simply observing a rise in the amount of lines of code (LOC) cannot be correlated with a rise in the defect rate. Classic software metrics such as the McCabe complexity are used alongside a set of object oriented metrics as exemplarily described in Section 2.4. It is also strongly highlighted that there is no set of metrics that will fit all software projects and that the metrics which are applied have to be chosen on a per-project basis. Also, if the measurement of metrics is already an integral part of the software development process this method cannot produce valuable insights.

Besides measuring metrics they also examine how code churn reflects in the rate of defects inside a software system [17]. In their approach not the absolute code churn is used to reason about defect rates, but rather relative code churn measures are used. These include amongst others

- LOC churned / total LOC and
- files churned / total files.

They show how an increase in those numbers often times correlates with an increase in the defect rate of the software component at hand. That way defect prone components inside a system can be identified.

Nagappan et al. show that not only metrics but also way simpler measures such as relative code churn can be correlated with defect rates. However, this approach also does not take the time factor into account.

5.2 Aggregating Code Metrics

Serebrenik et. al [33, 34] perform an evaluation of different aggregation mechanisms and highlight which ones show a high correlation. They also confirm that bare LOC are

not a relevant metric as also stated in [18]. Aggregating metric values is necessary as most metrics are defined on a micro level such as functions or classes. However, conclusions will mostly be drawn on a component or system level. Their work indicates that the correlation of metrics and defects strongly depends on the aggregation mechanism used to gather a unified value for a software system.

Mordal et. al [35] show that a typical approach to aggregate software metrics is computing the average of all individual values. Unfortunately this comes with some undesirable side effects as it smoothes the result set. This way it dilutes the impact of bad results in the overall quality. In [35] the Squale model described in Section 3.3.1 is introduced. This model was designed with the ISO 9126 [23] standard in mind. To effectively compare different metric values they are first normalised into a given interval of values. This interval is continuous and also has finite bounds. These properties make comparison of different values much easier.

Another very important fact that is stated, is how incorporating metric measurements into the development cycle can have an effect on the individual developer performance. In one case developers began to only chose work packages which would benefit their personal "reputation". This means developers shifted from trying to improve the software to improving how they looked from a metrics perspective. If these effects occur, management must intervene as the sensible reason why metrics are implied seem to not have been made clear in the organisation.

5.3 Determining Experts

Eyolfson et. al [40] show how time of day and developer expertise influence the amount of bugs that are produced during software development. Especially if developers work late into the night they are more likely to produce errors in the code they write. This can most certainly be traced back to the fact that they wear out over the day and therefore are less able to concentrate on the work they are doing. Especially in the hours between midnight and four o'clock in the morning the amount of bugs relative to the amount of commits is extremely high. It is also found that developers with more expertise produce less bugs. However, expertise is only defined by the number of commits a developer has contributed to a code repository. Still, this information seems to be adequate in order to reach a certain amount of certainty to decide whether a developer can be considered an expert or not.

We found this insight to be very useful as we could use it as a first estimation to pick a likely candidate to be an expert. What Eyolfson et. al do not consider is the quality of

the committed files. We want to refine this model and enrich it with information gathered from code quality measures in order to better determine experts. This should be very important when developers have worked for a long time on a project. For example it is apparent that a developer who has committed one thousand revisions has more expertise than a developer who only contributed one hundred revisions, simply because he has worked longer with the code and probably already knows his way around. That premise does not hold anymore if both developers have committed a considerable amount of revisions. Then both developers probably have deep insights into the source code and other factors, such as recency should be considered in order to tell which one of them is the current expert.

Schuler et. al [41] show how expertise of developers can be retrieved from version archives using static code analysis. They point out that there are two kinds of experts – *implementation experts* and *usage experts*. The first group are the developers who actually write the code. This group can be found by simply extracting which developers are responsible for changes in certain files. Finding the second group is a little harder. Developers who are usage experts use API methods that were written by others. They therefore have a deep understanding on how to work with existing code or third party libraries.

Still, this approach does not take the quality of the code into account, neither for implementation experts nor usage experts. In this thesis we will focus on implementation experts and further refine the method to determine this kind of experts using code quality metrics. Additionally we are also able to verify our findings. Even though the work of Schuler et. al presents a means to identify both implementation and usage experts it lacks an evaluation of the results.

LaTozza et. al [42] show different kinds of expertise. While expertise can mean that one person knows certain facts others do not know, expertise can also be knowing where to look if you do not know the fact; In other words knowing where the documentation for a certain problem is. The problem here is, that in a lot of organisations documentation is outdated the moment it is written and expertise manifests in the minds of the developers when they work in the organisation long enough. Once a developer does not need the documentation for something anymore he or she also will not update it. This is essentially critical if new members join a team.

As they cannot have the complete knowledge of the software from day one and they also cannot rely on the documentation which is present in most companies, an experienced mentor is usually assigned to new developers. Mentors are the designated point to go

to if the new developer has any questions. However, even the mentor might not always know the right person to ask or his or her knowledge might be out of date. If a certain person, who could have been considered an expert at the given time the mentor last worked with the piece of code in question, has already left the company, the mentor probably also does not know whom to ask.

In this case multiple developers have to be interrupted in their work in order to get the needed answers. According to [42] interruptions by colleagues are ranked second when it comes to what hinders developers in doing their work. After they have been interrupted they must remember goals, decisions, hypotheses, and interpretations from the task they were working on, and risk inserting bugs if they misremember.

We therefore think that mentors can essentially benefit from the automatic expert determination we propose. When working properly it will reduce the amount of time needed to find experts and also the amount of developers that need to be interrupted.

Chapter 6

Findings

In this chapter we want to present our findings regarding the research questions we posed in Section 1.2. The data presented here has been gathered using the analyzr framework as shown in Chapter 3. We will evaluate the results using Signavio as the main use case as we have insights into the structure of the company and the developers have agreed to help us evaluate our findings. If applicable, we will also take the data we gathered on the jQuery or Eclipse JDT repository into account. For question Q4 which will be discussed in Section 6.4 we will also use a survey we did at Signavio to confirm or refute our findings.

6.1 Finding the Main Contributors

Knowing the main contributors of a software project is crucial knowledge. The code these developers committed over time forms a critical mass when it comes to the maintainability of the overall system. A big question we had to answer was how to decide who a main contributor is and who is not. In order for us to determine the impact of a developer we first need to define the following sets.

$$R = \text{The set of all revisions,}$$
$$F = \text{The set of all files, and}$$
$$A = \text{The set of all authors.}$$

We also define the two functions "author" and "date" as:

$$\text{author} : (R \cup F) \to A$$

date : a function that assigns a date to a revision $r \in R$ or a file $f \in F$

As we further want to limit the search space using the information we have considering time and date of certain revisions, we define two subsets R_t and $R_t(a)$ as follows.

$$R_t = \{r \mid r \in R \wedge \text{date}(r) \geq t\}$$
$$R_t(a) = \{r \mid r \in R_t \wedge a \in A \wedge \text{author}(r) = a\}$$

R_t is the set of revisions that have been committed on the date t or later. $R_t(a)$ further refines the set R_t to only include revisions contributed by a certain author a. Using these sets we can now compute the impact of an author a starting from a point t in time till the present day to be:

$$i_t(a) = \frac{|R_t(a)|}{|R_t|}$$

The work of Bird et. al suggests that developers whose impact attributes to 5% or more of the overall commits on a repository can be classified as main developers [43]. So a function $\text{md}_t(a)$ that decides wether or not a developer can be considered a main developer for a certain range of time can be defined as:

$$\text{md}_t(a) = \begin{cases} true & \text{if } i_t(a) \geq 5\% \\ false & \text{otherwise} \end{cases}$$

Whether or not this method delivers the correct results depends profoundly on how t is chosen. If t lies too far in the past developers who contributed a lot in the past but are not active anymore are also considered main developers. Even worse, taking their commits into account dilutes the scores of new developers who might be way more active on the repository right now but did simply not yet have the time to contribute a lot of code. On the other side, if we choose t to be too close to the current day, we neglect the work which has been done in the past. This results in a short sighted view of the current situation where people who might not have much experience but recently contributed a lot of code are considered main influencers.

In the following sections we first want to highlight some threats to the validity of our results and then show how this approach performs with different time frames using the repositories of Signavio and the jQuery project. For the analysis of our results we chose three time frames.

- The complete history of the repository,
- The past 62 days, and
- The past 31 days.

For each time frame we will show the main developers we determined and evaluate if the results can be considered valid.

At this point we are not able to evaluate the results for the Eclipse JDT as there exists no list of developers we could compare our findings to. Nevertheless we argue that the results would be similar to the ones of the Signavio and jQuery repositories as the results we draw from these repositories have been verified by the developers themselves in the case of Signavio or using the list of current developers in the case of jQuery.

6.1.1 Threats to Validity

One of the main threats to the validity of our results is the fact that the credentials of developers tend to change from time to time. A post-analysis of the data revealed that duplicates can be found amongst the authors. In most cases this happens if the username a developer uses for the SCM system changes. We are not able to detect these changes automatically. In a manual run through the authors we tried to consolidate the data as best as we could in order to get reliable results. However, we cannot be sure that we have found all duplicates. Also we only gathered data on the main branch of each repository. For Git repositories we used the *master* branch and for SVN repositories we used the *trunk*. Therefore developers who mainly work in branches are also not included in our analysis or do not show up as main contributors even though they might be very active.

Another point which could falsify our results is how often developers commit their code. SCM systems like SVN encourage developers to only commit their changes to the code when they have reached a certain amount of certainty that they do not introduce new bugs into the source code. But some developers still split their commits into smaller ones so that the changes for different components are also in different commits. If, on the other hand, Git is used as a means of version control one would expect a lot more commits as they are not uploaded to the repository right away but stored on the machine of the individual developer. However, once they are uploaded to the repository each commit counts. Therefore developers who just tend to commit more often than others would be ranked higher in the statistic we have chosen.

Figure 6.1 shows the results concerning commit behaviour of the survey we have done at Signavio. Even though the majority of developers does not commit more often than three times a day on average a small percentage commits with a higher frequency. Further studies on this matter should evaluate different approaches on how to tackle this problem.

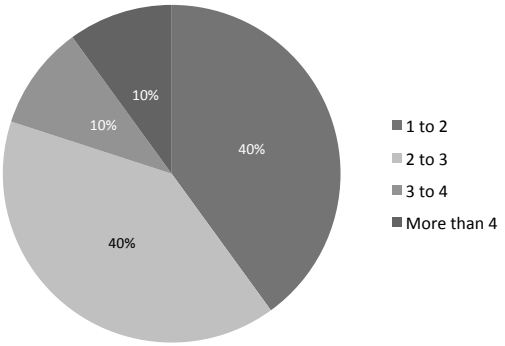


FIGURE 6.1: The commit behaviour of developers at Signavio differs in the amount of commits they produce on average per day.

6.1.2 Signavio

Figure 6.2 shows the impact of all developers who have ever committed files to the Signavio repository regardless whether they are currently working at the company or not. We can see that 7 of 37 developers can be classified as the main contributors to the repository. In this case all of the seven main contributors still work in the company. However, not all of them are still part of the core development team as the roles of them have changed from development to customer relations or tasks related with management.

In Figure 6.3 we can already see how the statistic changes if we narrow the time frame. If we look at a time frame of 62 days, for example Hester Marler and Annabelle Prudhomme who ranked second and fourth in the overall statistic are no longer considered main contributors. On the other hand Shawn Mayberry who only ranked 18th is now part of the group of main contributors.

This illustrates the importance of looking at the data from different perspectives. Even though the main contributors from Figure 6.2 might still have a broader understanding of the software system as a whole because they worked longer at the company and therefore also contributed more code to different parts of the system, the main contributors shown in Figure 6.3 are likely to have a better understanding of the current developments.

To drive this approach even further we narrowed down the time frame to 31 days in Figure 6.4. What we can observe here is that a lot of developers only miss the 5% hurdle by a very small amount. Especially the middle part with values ranging from 3% to 5% includes a lot of developers that were active during the time but are not considered main developers. This is also a result of the above-average amount of commits that have been contributed by Jadwiga Gillette. It seems as if 31 days is a too narrow time frame when it comes to deciding who the main contributors are or that the barrier of 5% has to be lowered in order to capture all developers.

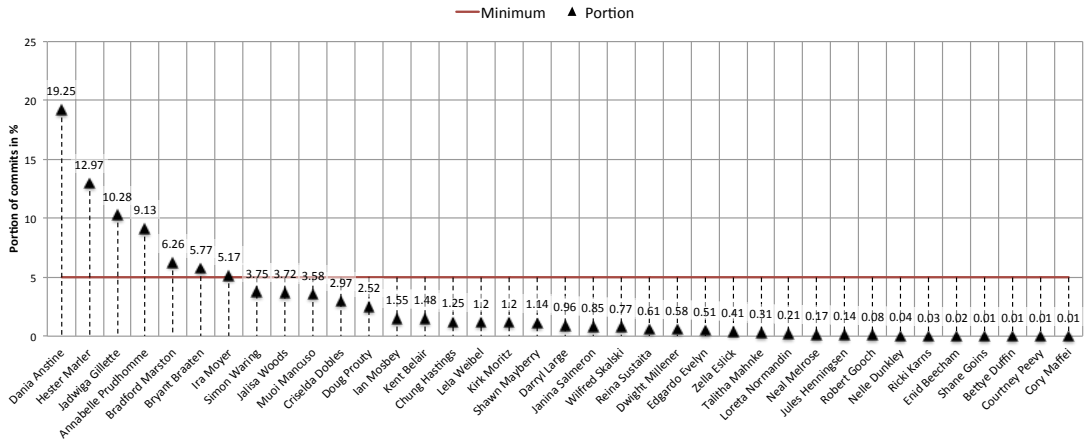


FIGURE 6.2: Contributors to the Signavio repository between April 4, 2009 and March 3, 2014.

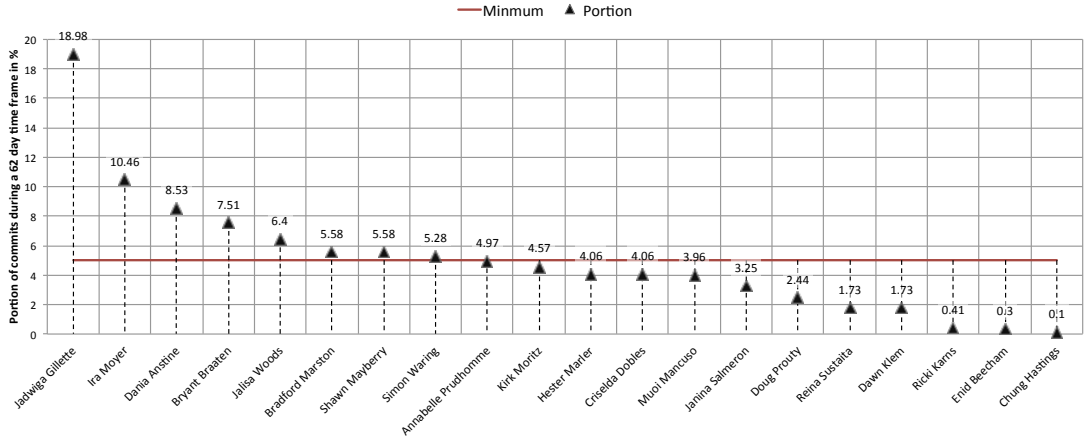


FIGURE 6.3: Contributors to the Signavio repository between January 3, 2013 and March 3, 2014.

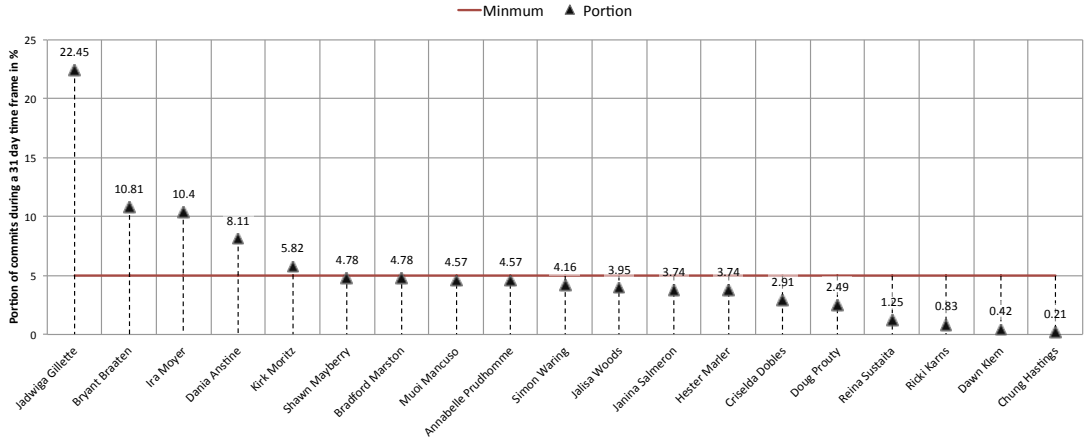


FIGURE 6.4: Contributors to the Signavio repository between February 3, 2014 and March 3, 2014.

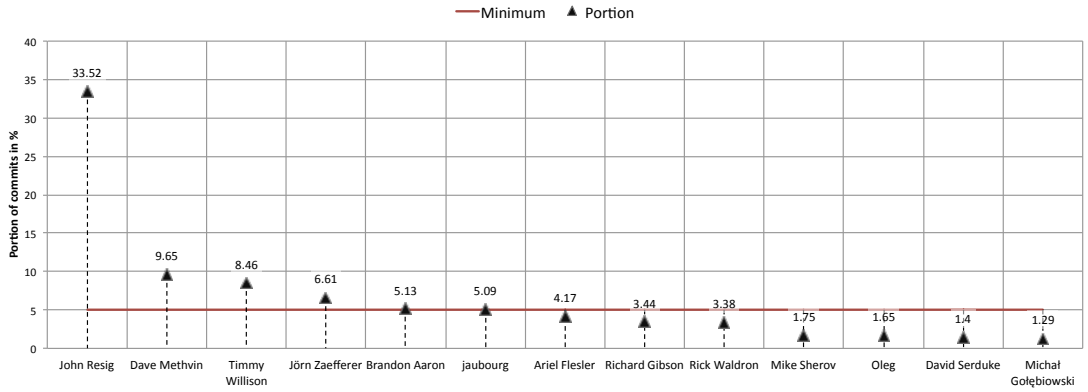


FIGURE 6.5: Contributors to the jQuery repository between March 22, 2006 and February 14, 2014. For reasons of readability contributors that amounted to less than one percent of the overall commits are not shown in this statistic.

6.1.3 jQuery

In order to assess whether our analysis shows the appropriate developers or not we will use the official staff-list for jQuery which can be found at `https://jquery.org/team/`. As for Signavio we will first look at all revisions ever committed to the jQuery repository and then narrow down the time frame.

Figure 6.5 shows a great deal of developers who have contributed code to the framework. For reasons of readability we have excluded developers who amounted for less than 1% of the overall revisions. The founder of the jQuery project, John Resig, stands out as till today he can be attributed to have committed 33.52% of the overall revisions. Nevertheless, even though they have contributed a lot to the project, John Resig and Brandon Aaron for example are no longer active contributors as can be seen on the team web page. John Resig now helps set the overall direction of jQuery and Brandon Aaron has left the team of developers.

When we have a look at Figure 6.6 Michał Gołębiowski who ranked last in the overall statistic has now become the leading contributor. We can also see that all developers who qualify to be main contributors are listed on the team page. Also Dave Methvin still contributes a lot of code to the repository while being part of the board of directors. This differs from what we have observed at Signavio where senior developers seem to more and more coordinate and plan work instead of actively contributing code.

Again, as we narrow the time frame even further as shown in Figure 6.7, we can see that Michał Gołębiowski, Oleg Gaidarenko, Timmy Willison, and Rick Waldron have been most active between January 14, 2014 and Feburary 14, 2014. Except Rick Waldron all of these developers are part of the jQuery development team. Rick Waldron is also part of the board of directors and supervises the jQuery core development.

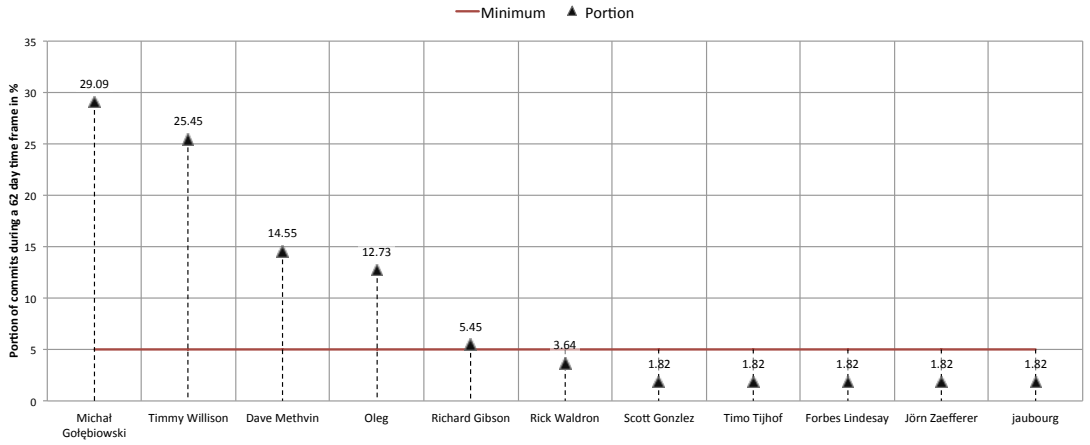


FIGURE 6.6: Contributors to the jQuery repository between December 14, 2013 and February 14, 2014.

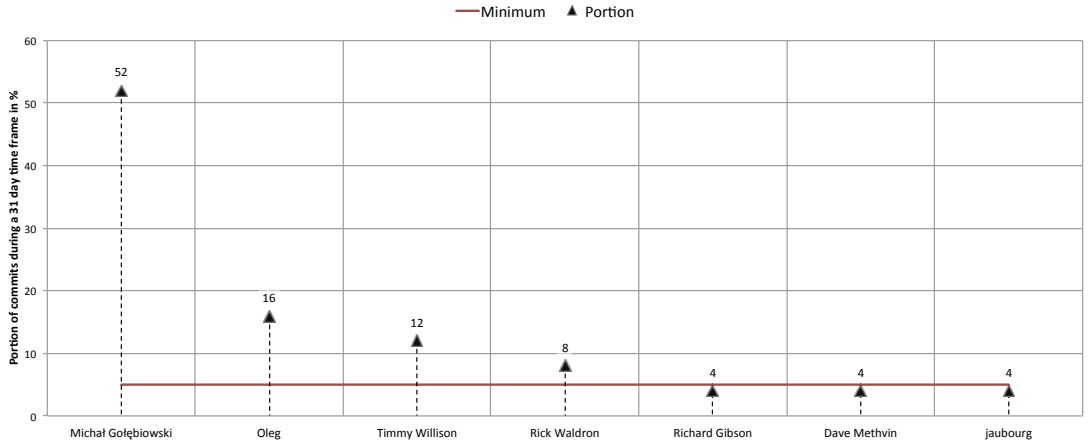


FIGURE 6.7: Contributors to the jQuery repository between January 14, 2014 and February 14, 2014.

6.1.4 Evaluation

We have shown that when it comes to determining the main contributors of a software project one cannot simply use the revision data for the complete history of a project. While this data might still be valuable as it shows which developers have a lot of experience regarding the systems as a whole it does not give insights into who currently works a lot with the code.

In our observation a time frame of around 62 days delivers the best results. This period seems long enough to not only focus on developers who just recently contributed code to a repository and short enough to exclude developers who are no longer actively involved in the development of the software or have left the organisation.

Even though the 5% border sometimes seems to be too sharp as it removed developers who have also contributed a reasonable amount of code, it shifts the focus to a smaller and more concise group.

We also considered an approach where we would not rely on the committed revisions but on the committed files, but we ruled this approach out, as it was too uncertain whether the results would really differ. Also, we would than have to differentiate between programming languages. For example as Java enforces one class per file (disregarding inner classes) a lot more files have to be created when developing in this language as for example when writing JavaScript code. This would mean that main contributors would have to be determined on a per-language level which might be interesting in future work but was out of the scope for this thesis.

6.2 Classification of Developers

Most software products can be split into two major parts. The users mostly interact with a custom build frontend whose main purpose is to present an interface that is easy to use and that displays the results of any user interaction. On the other side is the backend which performs data related tasks and calculates results that will be presented in the frontend.

As the tasks of both parts differ a lot also the languages which evolved to solve them differ. In the web languages such as Java, Python, or Ruby have become very popular for backend related tasks. On the contrary, JavaScript has become the de facto standard for client side scripting.

In our approach we assume well defined sets of languages which can be correlated with the respective part of the application. We therefore define the following sets.

$$\mathcal{T}_F = \text{The set of languages, which belong to the frontend, and}$$
$$\mathcal{T}_B = \text{The set of languages, which belong to the backend}$$

Furthermore as we want to classify certain developers we define the following subsets.

$$F(a) = \{f \mid f \in F \wedge a \in A \wedge \text{author}(f) = a\}$$
$$\mathcal{T}_F(a) = \{f \mid f \in F(a) \wedge \text{type}(f) \in \mathcal{T}_F\}$$
$$\mathcal{T}_B(a) = \{f \mid f \in F(a) \wedge \text{type}(f) \in \mathcal{T}_B\}$$

$F(a)$ is the set of all files a developer a has contributed to a given repository. $\mathcal{T}_F(a)$ and $\mathcal{T}_B(a)$ are the sets which contain files that were added or modified by author a with

types that correlate to the frontend or the backend of a software system respectively. The method type : $F \to (\mathcal{T}_F \cup \mathcal{T}_B)$ is used to assign a type to a file f. Already now we can see that $\mathcal{T}_F$ and $\mathcal{T}_B$ cannot be defined universally but have to be defined on a per-project basis.

As even frontend developers might have to implement some functionality in the backend and vice versa the classification will not be binary but expressed as a probability. We define $p_F(a)$ and $p_B(a)$ as the probabilities that a given developer a is a frontend or a backend developer as follows.

$$p_F(a) = \frac{|\mathcal{T}_F(a)|}{|F(a)|} \qquad \text{and} \qquad p_B(a) = \frac{|\mathcal{T}_B(a)|}{|F(a)|}$$

For the results presented in this chapter we assume a developer a to belong to a certain category if either p_F or p_B exceeds a value of 50%.

Figure 6.8 shows the classification of all developers at Signavio. An aggregated version is shown in Figure 6.9a. These are the total numbers considering all developers which have ever worked at Signavio. As the SCM system checks each commit for the correct format of the commit message and also a continuous integration (CI) tool is used at Signavio, two non-human authors would also be part of this statistic. A test user which is used when new rules for the SCM system are implemented and tested, and a user for the CI tool that commits configurations. Both these users have been removed in order to focus on the real developers.

As one can see in Figure 6.8 most developers have a strong relation to either the frontend or the backend. However, some developers show a fair amount of activity on both sides. These authors are especially relevant if staff shortages occur. We consider developers who have a probability p over 20% to be also able to work in the contrary part if necessary. The results of this analysis are shown in Figure 6.9b. For Signavio 1/7 of the frontend developers could also work in the backend and 1/5 of the backend developers could also work in the frontend.

Unfortunately this analysis does not take the fact into account that developers leave the company. If we would solely rely on this data we would always have an outdated view of the company at hand. We therefore decided to focus on a more recent period of time.

Figure 6.10 shows the developers and their respective classification for a time frame between February 3, 2014 and March 3, 2014. Again, non-human authors have been removed from this statistic. The first thing we see is that the amount of developers has decreased from 34 shown in Figure 6.8 to 18 in Figure 6.10. This fact also manifests in the ratio between frontend and backend developers. In Figure 6.11a we see that there

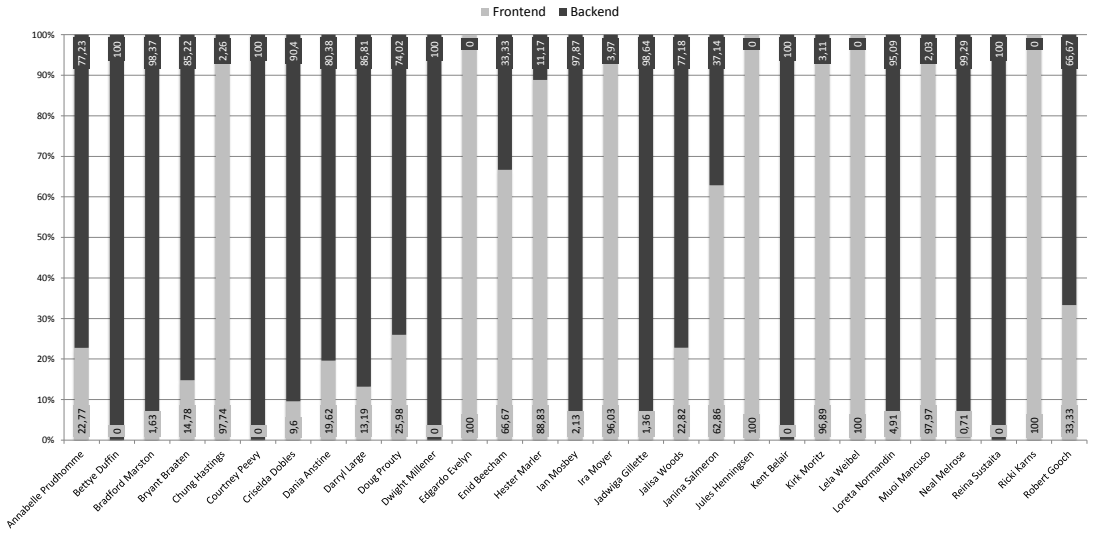
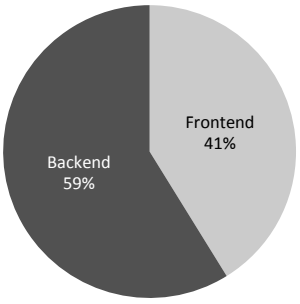
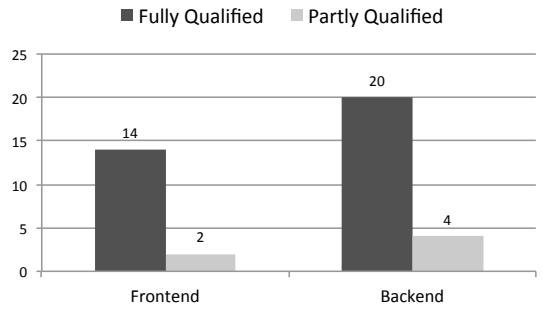


FIGURE 6.8: Overview showing how much each developer at Signavio can be classified as a frontend or backend developer considering all files ever committed.



(A) Percentage of frontend and backend developers at Signavio.



(B) Amount of fronend and backend developers who could also be exchanged if necessary.

FIGURE 6.9: The classification of developers can also be used in order to find frontend developers who can help out in the backend and vice versa considering all files ever committed.

is an equal number of frontend and backend developers as opposed to Figure 6.9a which would suggest, that more backend developers work in the company. We also see that only 1/9 of the frontend developers could work on the backend, but 1/3 of the backend developers might be able to also tackle frontend tasks.

We argue that management decisions that are concerned with staffing should always use the most recent data in order to get valuable insights into the current staff situation. This does not mean that the classification of developers should be based on a limited set of revisions, but that the set of developers under consideration should be reduced to the ones, who have been active in the recent time. Otherwise also developers who do not work in the organisation anymore would be part of the analysis and therefore falsify the overall picture of the amount of development resources that are currently

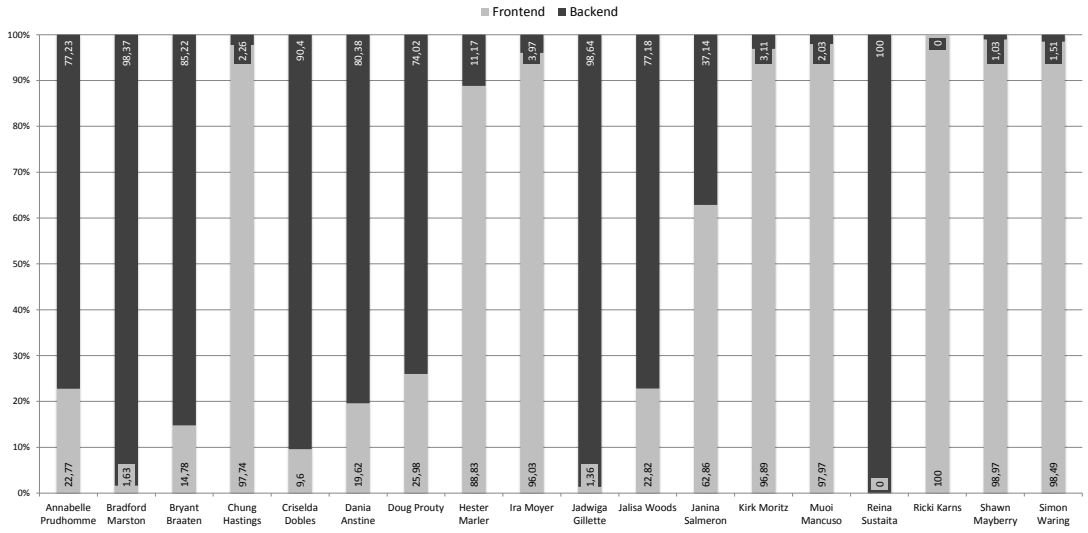
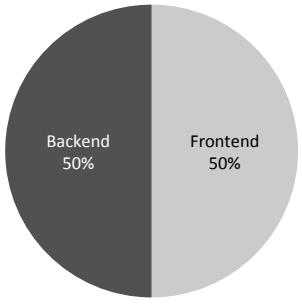
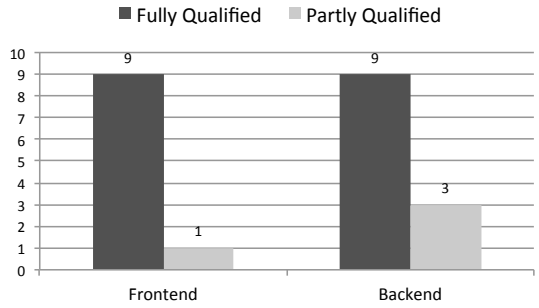


FIGURE 6.10: Overview showing how much developers who committed files between February 3, 2014 and March 3, 2014 can be classified as a frontend or backend developer.



(A) Percentage of frontend and backend developers at Signavio



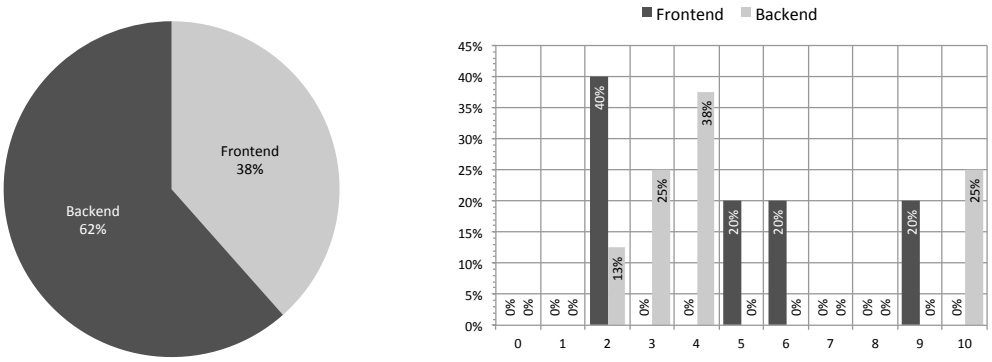
(B) Amount of frontend and backend developers who could also be exchanged if necessary

FIGURE 6.11: Classification of developers who committed files between February 3, 2014 and March 3, 2014.

available. Statistics like these help to understand how developers work and also enable project leaders to better plan projects as they can understand how resources can be best allocated.

6.2.1 Threats to Validity

As our approach is bound to examining which languages are used and how much different developers work with them, we cannot distinguish between different kinds of developers, if the same language is used to write both the frontend and the backend. This might become an issue in the future as for example JavaScript has also found its way to the server side with, for example, nodeJS. In that case new means to differentiate between backend and frontend code must be used. We could think of a simple code analysis



(A) Classification as done by the Signavio developers.

(B) Developers estimate if they could also work in the contrary part of the application.

FIGURE 6.12: Classification of developers as a result of the evaluation at Signavio.

looking at which packages are used by developers but clearly more research into this topic is needed in order to draw insightful conclusions on the matter.

6.2.2 Evaluation

To evaluate our findings we had the developers at Signavio state if they are concerned with the backend or the frontend and also rate if they think they would be able to work in the contrary part of the software. The results of this survey are shown in the Figures 6.12a and 6.12b.

As shown in Figure 6.12a 62% of the developers stated to work in the backend, whereas 38% stated to work in the frontend. Those numbers do not align with our findings presented in Figure 6.11a. We were able to determine that this discrepancy is the result of two developers leaving the company only days before the survey. Another developer just went on a holiday and was therefore not able to take part in the survey.

However, this only emphasises the point we want to make in this section which is that data can change in a very fast manner and that analysis should not be based on data which is outdated. Even though we limited our analysis to the last month we were not able to observe those changes in the staff.

6.3 How Developers Evolve

As stated in research question Q3 in Section 1.2 we want to evaluate if code metrics can be used in order to assess whether a developer has evolved over time. The assumption we make is that developers who work longer on a software project also write code which

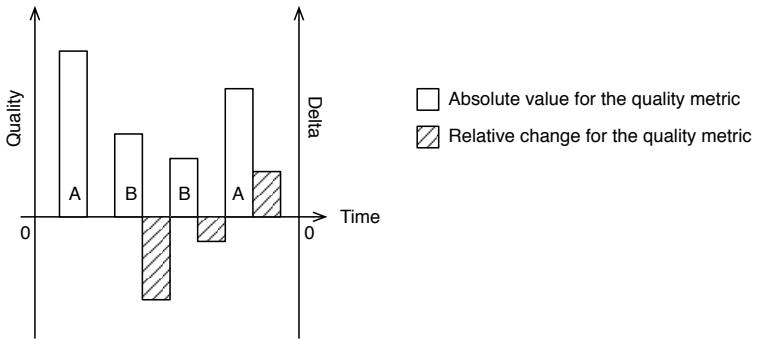


FIGURE 6.13: Two developers and the delta values they produce when they commit a file to a software repository.

is less complex and easier to understand as they know how the different components interact with each other and are also able to reduce the amount of code duplication by employing reusable components. This should then reflect in the software metrics we measure. The graphs we will present in the next sections can be understood as follows.

Each graph shows multiple curves in different colours where each colour represents one code metric. Which code metric can be associated with which colour is depicted on the different y-axes on the graphs. As we have transformed the individual values for the different metrics, to comply to a normed co-domain, using Squale as described in Section 3.3.1 higher values stand for better code quality regarding the metrics. We should also note that the different graphs do not show the actual values for the software metrics, but the changes which have been recorded. Therefore the range of the values on the y-axes is not necessarily the important part but the trend which the graphs show. If a lot of files are changed, then also a lot of deltas can be recorded and therefore the value depicted by the graph will be higher or lower accordingly. Relying solely on the absolute values would result in graphs which are of no use to us. We want to highlight this fact with a small example depicted in Figure 6.13.

Given two developers A and B. Developer A creates the initial revision of a file and chooses a very good design with small methods and a low complexity. After A has pushed his changes to the repository B changes the file in a way so that the overall quality significantly decreases. A notices this and decides to react. After a thorough refactoring A has managed to reduce the complexity of the code to a point which is not as good as in the initial commit but way better than what B accomplished to do.

If we would rely on the absolute metric values it would look like A increased the complexity of the file as his second value for the code metrics is lower than his first. However, this is not true as B was the one to blame for the decrease of quality which reflects in the metrics and even though A did not manage to restore the initial value his contributions

resulted in code with a better quality. We therefore concentrate on the deltas the two developers have caused. Now it is clear that B has decreased the quality as his deltas are negative and A managed to reduce the complexity of the code as his deltas are positive.

In this section we will present our findings regarding how metric values behave over time. We utilise the use cases we have presented in Chapter 4 and evaluate our results by examining the software repositories as a whole and by focusing on certain developers in particular.

6.3.1 Threats to Validity

A problem which should be addressed in future work is that some revisions carry information that dilutes the results but which is hard to detect. As Signavio uses SVN, revisions that represent a merge can potentially be problematic. When a developer merges one branch into another then the revision he or she creates accounts for all changes made to all files which have been modified in the branch. This in turn means that this developer is made accountable for all improvements and declines in the quality of the changed files. However, this does not reflect the work of said developer as he or she might not have changed any of the files, but simply is responsible for merges. We therefore argue that future work should pay attention to merge revisions and how they can be detected and excluded from the analysis.

After we ran our analysis, we noticed some irregularities in our data set which we further examined. We realised that we had to reduce the set of revisions that are used in the analysis because of the following reasons.

6.3.1.1 Unrealistic Positive Spikes of the Delta Values

When we had a closer look at the spikes in our graphs, it became clear that files, which have been added to the repository almost always cause an increase of the metrics. This is due to the fact that the default value for the different metrics in our data model is zero and therefore any value, which is greater than zero, will cause positive delta values. We compute metrics per function, but aggregate them using Squale into one value for each metric before we save them. Therefore all values are bound to a range between zero and three which circumvents any negative deltas. As we still need those revisions to set the base value for the metrics, we cannot simply ignore them at all. We only consider files that do not have a change type of “add” during the analysis. The next time this file is pushed to the repository, after it has been changed, the deltas can be computed using the base value for the metrics and thus have the correct value. Even though we loose

information because we neglect the initial "quality" of a file, we were able to enhance the validity of the overall result of our analysis.

6.3.1.2 Unrealistic Negative Spikes of the Delta Values

This is the counter part to the positive spikes which occurred when new files are added to the repository. As deleted files can no longer be measured the value for all metrics will remain at the default value which is zero. Therefore the value of all deltas that are computed using the difference between the current value for a metric and the value the metric had in the last revision would drop significantly. Again, simply ignoring files with a change type of "delete" is not an option. This kind of information is used in order to compute which files are currently present in the repository without doing any operations on the file system. This is done by concatenating the change type which has been assigned to each file over time. If the resulting string does not end with a "D", representing a delete operation, then this file is currently available. When we select repositories for our analysis we use this mechanism to quickly assess whether the amount of files we are able to measure is high enough to make the analysis of the data beneficial. Information regarding deleted files is therefore kept in our system, but is ignored when we perform any analysis that deals with the change of metric values.

6.3.1.3 Corrupt or Erroneous Files in the Repository

After we excluded added and deleted files from our analysis, we could still see some spikes (now only positive ones) of the delta values. We examined the revisions to which the files belong to and also the files themselves but could not find any indicators regarding the origin of the spikes. However, we were able to find the reason when we looked at the previous version of the files. If the tools we use to measure the quality metrics encounter a file which they cannot process because it contains errors, they simply skip it. When we process the results of the tools we are missing the files that contain errors and therefore no metric values are assigned to them. This does not influence the deltas as they remain at zero. However, if the next version of a faulty file is measured and the results are stored, the deltas spike to the current value for each metric and thus we see spikes in the data. Our framework now tracks the status of files, so if no results are reported for a given file, this file is marked to be erroneous and is ignored during the analysis. One problem of doing so is that we might not find a predecessor for a file. When this happens we would face a scenario similar to the one where we had to deal with newly added files. As no previous version could be found and the change type of this file is not "add" the deltas would spike up to the current value of the metric. In this case we trick the system

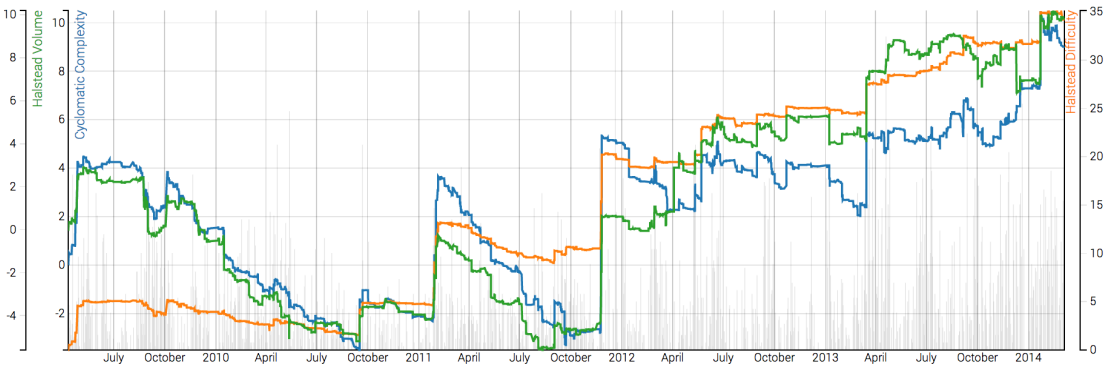


FIGURE 6.14: Development of the complexity metrics for the frontend of Signavio.

by changing the change type of said file to be "add". This way we keep track of the file and also define a base value for the metrics.

We have shown how basic errors in the implementation of our framework could have falsified the results of our analysis and thus also the conclusions we draw based on them. While we are certain that we have reached a degree of stability and quality in the framework to present our results, we cannot prove that we have eliminated all error sources.

6.3.2 Signavio

At Signavio especially the metrics for the frontend code behave as one would expect in a typical startup. In the beginning a lot of work went into quickly growing the application in order to make it viable as a product. This also means that not much time could be spent refactoring the code. The metrics in Figure 6.14 reflect that behaviour. In the first two years the metrics and thus the quality of the code mostly decrease and only rise at certain points when a refactoring needed to take place. When the company left the critical phase and new features did no longer have to be implemented in a very short amount of time, developers could focus more on the maintainability of the code. A new development method and more and more integration tests in the frontend support this trend. Especially during the last year much effort has gone into reducing code duplication and focussing on building reusable components. This in turn reduces the Halstead volume as fewer lines of code are needed in order to implement new functionality and also the cyclomatic complexity as checks only have to be performed in the unified components and not repetitively throughout the whole codebase.

The efforts which have been undertaken in the frontend to write code which is easier to maintain and understand started earlier in the backend. This can be seen in Figure 6.15 where already starting at the end of 2011 the metrics indicate a drastic increase

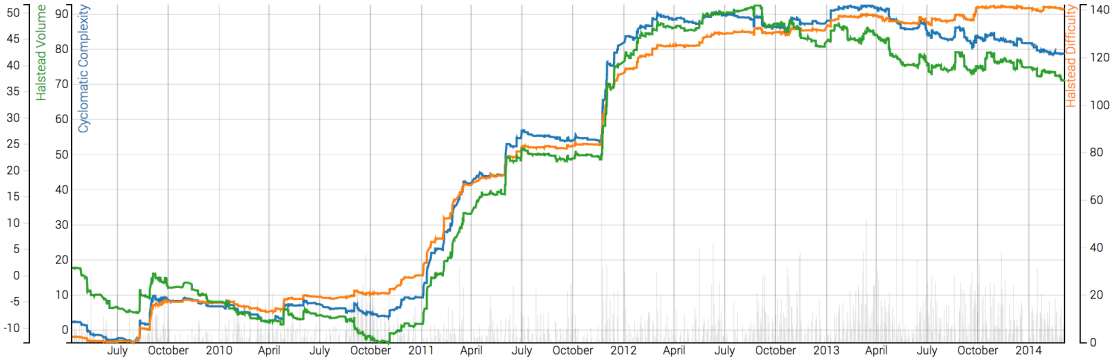


FIGURE 6.15: Development of the complexity metrics for the backend of Signavio.

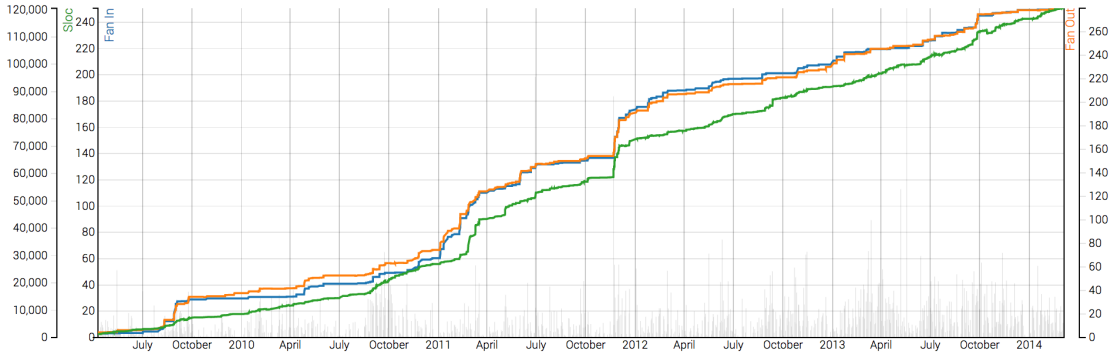


FIGURE 6.16: Development of the structural metrics for the backend of Signavio.

concerning code quality. However, after a lot of components have been refactored we currently see a slight decrease concerning the code metrics. We can use this information in order to show developers that once again their focus should shift from implementing new features to keeping the code maintainable. Albeit this current decrease we have seen enough increases during the past years in order to make the assumption that the overall code quality of the backend code for Signavio has improved over time and is now in a state where it is both easy to maintain and also easy to understand as the complexity is kept to a minimum.

Figure 6.16 led us to the conclusion that the classification we found and used for structural metrics such as C_e and C_a as described in Section 5.2 are too loose. We can see that the values for C_e and C_a always increase and never decrease. They also strongly correlate with the SLOC metric which means that more lines of code (e.g. new classes or methods) result in an increase in said metrics. Of course, this effectively renders them useless for us and future work should refine the scale by which the values for C_e and C_a are transposed to the Squale model.

After we have looked at the Signavio repository as a whole we now want to show how the curves change if we look at certain developers in particular. We chose the current main contributor of the backend Jadwiga Gillette. The graph which corresponds to the

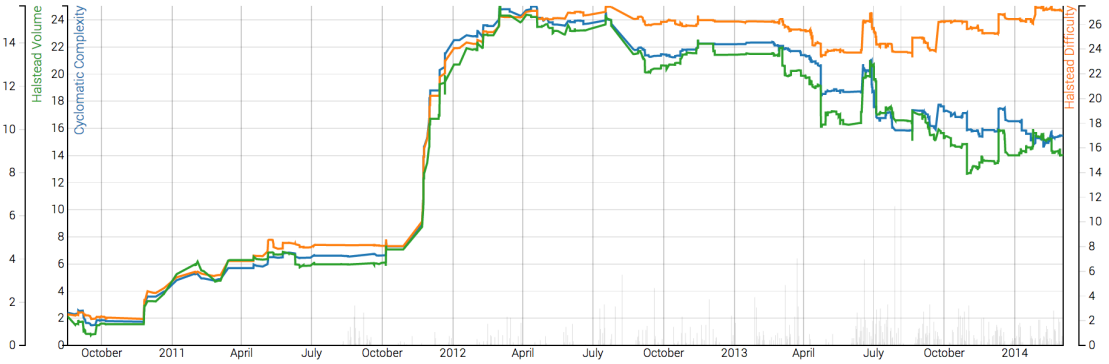


FIGURE 6.17: Development of complexity metric deltas for Jadwiga Gillette.

changes he created is shown in Figure 6.17. We see only few improvements during the first year the developer has worked at the company. When comparing this with other developers this initial period is sometimes longer, and sometimes shorter depending on how frequently the developer works at the company. This timespan represents the phase where developers become acquainted with the code. Full-time employees normally need not as much time as working students, as they not only spend more time working with the code but can also focus their work much better. After the developer knows how to work with the code we assumed he is able to dramatically improve the quality of the code he is working with. Still, right now we see a decreasing trend of the metrics. We therefore argue that expertise of a developer cannot be simply derived by, for example, summing up all delta values in order to retrieve one number describing the code quality which is produced by this developer. In fact, we think that the ratio between the number of increases and decreases a developer has caused yields better insights.

If a developer has once participated in a refactoring which led to an increase of the code metrics, but after that only decreased the quality, simply summing up all deltas might still convey the impression that this developer generally contributes less complex code. If we, however, count how often a developer increases quality and set this value into relation with how often he decreases the quality we get a better picture of his or her work habits. This way a developer has to constantly strive to better the code quality.

6.3.3 jQuery

In order to draw conclusions from the graph for the jQuery repository we use a list of major releases[1] in order to explain certain changes. We think it is interesting how accurate the descriptions for the releases reflect in the code metrics. A permanent decrease of the complexity metrics can be observed until the beginning of the year 2012. This is probably caused by growing the framework and gradually adopting to more and

[1] `http://en.wikipedia.org/wiki/Jquery#Release_history`

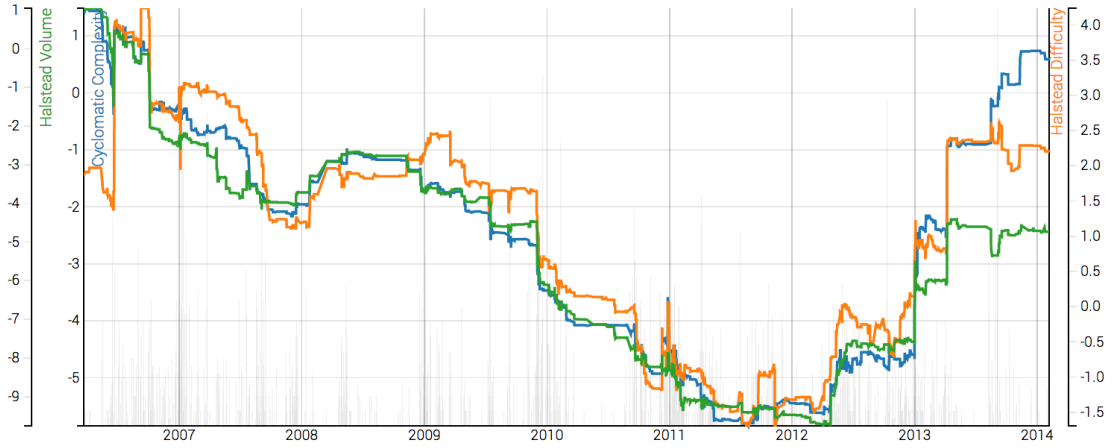


FIGURE 6.18: Development of the complexity metrics for jQuery.

more browser quirks without breaking the backwards compatibility. On August 9, 2012 jQuery version 1.8 has been released and a complete rewrite of the selector engine had been done in the weeks and months leading to this date. We can also observe an increase in the metrics leading to this date. Another major release was version 2.0 on April 18, 2013 which completely dropped the support for Internet Explorer versions six to eight. If we have a look at the graph around this time we can see how the values for both the cyclomatic complexity as well as the Halstead metrics spike up. This is exactly the kind of behaviour we expected when major parts of the framework that deal with the quirks of those browser versions are removed.

6.3.4 Eclipse JDT

The Eclipse JDT can be used in order to show a contrary example of how software development can be carried out. As shown in Figure 6.19 the complexity of the project has almost only increased over the past years. Only at the beginning of 2004 we can see a rise in the complexity metrics. This is most probably caused by the fact that the projects first open source release was on June 21, 2004 and the developers tried to clean up the code in order to make it easier for an open source community to form around the project. From this point on a lot of features and extensions have been added to the initial project but the metrics indicate that the developers did not care too much about writing code that is easy to understand and maintain. The metrics suggest that each new feature also introduced more complexity into the software. However, starting in 2008 the curve flattens which might suggest that features added after this point at least do not add as much complexity as have the features which have been added before that date. We would still suggest that future developers focus more on reducing the complexity of the source code to keep Eclipse a viable platform.

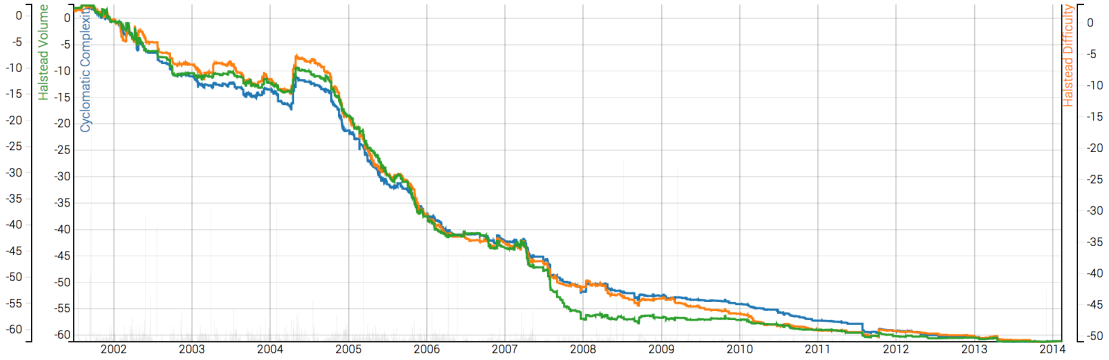


FIGURE 6.19: Development of the complexity metrics for the Eclipse JDT.

6.3.5 Evaluation

During our analysis we found that, even though the Halstead metrics and the McCabe complexity should not correlate, we can observe a similar trend for both curves in most of the graphs. This fact leads to the conclusion that developers will in most cases not only grow the amount of lines of code but are also likely to introduce path complexity as described in Section 2.2. We also observed that it seems to be easier to reduce the McCabe complexity than the Halstead Volume or the Halstead Difficulty when it comes to languages, such as Java, which have a more expressive syntax and therefore require more lines of code. For example, in Figures 6.15 and 6.17 we notice that even though efforts have been undertaken to reduce both kinds of complexities, the pace at which the cyclomatic complexity is reduced is almost twice as high as for the Halstead complexities. This, however, leads to the assumption that the classification we used in order to decide whether a value of a metric can be considered high or low should be changed if the language under consideration changes. Some languages require less statements per method on average because they are equipped with more so-called syntactic sugar. This essentially means that even though we have chosen metrics that do not favour certain languages, we still cannot completely abstract from the programming languages as we need to consider them, if we aggregate the values of a metric using models such as Squale (c.f. Section 3.3.1).

We also found that developers should not be rated based on the sheer amount of value changes in the metrics they account for. A measure should be applied which expresses the continuity with which they influence the complexity. Even if a developer has contributed a lot of code which significantly reduced the complexity of the files he or she was working on this should not serve as a cushion. A good developer should strive to constantly improve the overall system by keeping its complexity to a minimum and therefore making it better maintainable for others. Unfortunately, the complexity of a software system is no tangible factor for the individual developer, but perceived differently by each person

in the development team based on their expertise and the time they spent working with the system. To overcome this issue we propose a measure which incorporates the ratio of improvements to degradations of software metrics to tell whether a developer is helping to reduce the complexity of the overall system or not. However, this metric cannot be used in order to precisely assess whether the code of a developer is prone to error or not.

Concluding we can state, that we are able to assess if a developer evolves in a way where he or she strives to reduce the complexity of a software system. However, this measure cannot be used in order to assess whether an individual developer gains expertise. More factors, as for example shown in Section 6.1, should be used to refine our measure in order to make assumptions regarding expertise.

6.4 Expertise of Developers

After we have studied how time affects the selection of main contributors to a repository in Section 6.1 and also have shown how developers evolve in Section 6.3 we want to show how we determine experts for parts of a software. In order to do this we need to assign a score to each developer using the information we have on them. In Section 6.1 we solely considered revisions that were created in the repository *after* a certain point in time. For the score we want to compute, we need to use the whole history of a developer *up to* a certain point in time. We therefore define the sets R_T and $R_T(a)$ as follows, which should not be confused with R_t and $R_t(a)$ that were used in the last sections.

$$R_T = \{r \mid r \in R \wedge \text{date}(r) \leq T\}$$
$$R_T(a) = \{r \mid r \in R_T \wedge a \in A \wedge \text{author}(r) = a\}$$

R_T is the set of revisions that have been created up to T and $R_T(a)$ is the set of revisions that have been create by the author a till the date T. As we are now dealing with the measurements which have been computed for each revision, we also need to define

$$M = \text{the set of all measures,}$$
$$\text{measures} : R \to \mathcal{P}(M), \text{ and}$$
$$\text{delta} : M \to \{n \mid n \in \mathbb{R} \wedge -3 \leq n \leq 3\}$$

The function "measures" assigns a set of measures $m \in M$ to a revision. The method "delta" is used in order to retrieve the change in the value of a metric relative to the last revision. As stated in Section 6.3 we are not focussing on the actual value of a measurement, but on the changes a developer has caused. In order to state whether a

measure has increased or decreased we define the functions *in* and *de* respectively.

$$in(m) = \begin{cases} 1 & \text{if delta}(m) > 0 \\ 0 & \text{otherwise} \end{cases} \qquad \text{and} \qquad de(m) = \begin{cases} 1 & \text{if delta}(m) < 0 \\ 0 & \text{otherwise} \end{cases}$$

If a measure m has increased, $in(m)$ will yield 1 meaning that the software metrics indicate that the quality has increased. On the contrary, if a measure m has decreased $de(m)$ will yield 1, meaning that the software metrics indicate that the quality has decreased. To count all increases and decreases for all revisions $r \in R_T(a)$, we define two more functions $\text{increases}_T(a)$ and $\text{decreases}_T(a)$.

$$\text{increases}_T(a) = \sum_{r \in R_T(a)} \left(\sum_{m \in \text{measures}(r)} in(m) \right)$$

$$\text{decreases}_T(a) = \sum_{r \in R_T(a)} \left(\sum_{m \in \text{measures}(r)} de(m) \right)$$

We have learned that in general developers who work longer on a project are the more experienced ones. However, we argue that the experience which relates to the amount of commits a developer has created should not be a linearly function. Even though there must be a distinction between developers who only created a small amount of commits and the ones who contributed a lot, this distinction does not necessarily hold when developers have worked in the organisation for a longer period of time. In this case the amount of quality improvements they have caused in the source code should be used in order to distinguish who should be considered an expert and who should not. Therefore we define the function $\text{score}_T(a)$ that assigns a score to a developer at a certain point in time as:

$$\text{score}_T(a) = \begin{cases} \frac{\text{increases}_T(a)}{\text{decreases}_T(a)} \times \ln\left(1 + |R_T(a)|\right) & \text{if decreases}_T(a) > 0 \\ \ln\left(1 + |R_T(a)|\right) & \text{otherwise} \end{cases}$$

The factor of increases to decreases can only be computed if the author has already caused decreases of the software metrics. While we could have solely used the value of $\text{increases}_T(a)$ in this case, we noticed that this scenario only applies to developers who have contributed very few revisions. Using $\text{increases}_T(a)$ resulted in the fact that most of the new developers were selected as experts. This is obviously unreasonable and we therefore decided to fallback solely on the amount of revisions they have contributed to the repository in this case. Our approach uses the scores computed by this method but also does not neglect what we have learned in Section 6.1. Time is again a crucial factor. Thus only developers who were active during the last 31 days (i.e. the last month) in the

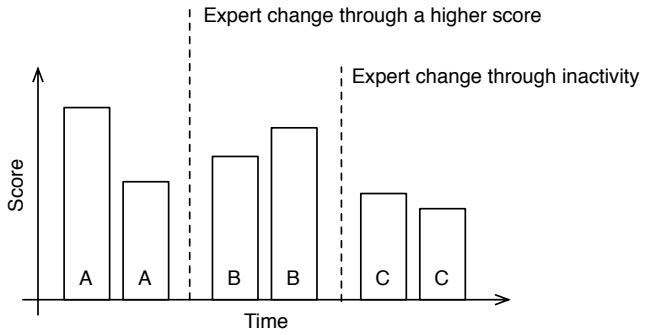


FIGURE 6.20: Experts can loose their status when their score is too low or if they become inactive.

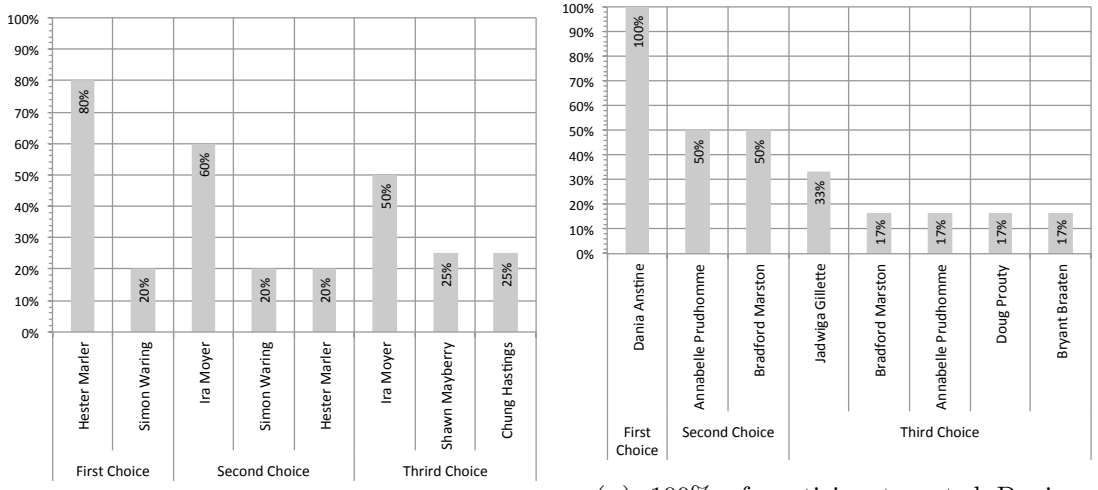
part of the software that is examined can gain the status of an expert. In Figure 6.20 we see three experts A, B, and C and their respective score during the time when they were the expert. No real-world values are assigned to the scores as this is only an abstract illustration of the principles behind our algorithm to select and expert. It highlights two possible ways the current expert can loose his or her status. In the first case, when B becomes expert instead of A, both developers A and B are currently active in the repository but the score of B surmounts the score of A. The seconds case highlights the fact where both A and B are inactive, which enables C to become the expert even though C has a lower score than B.

6.4.1 Survey

In order to evaluate the results our framework produces we conducted a survey at Signavio. The survey consisted of four parts. In the first part developers had to state if they either work in the frontend or the backend and also estimate how often they commit on average during one day. Based on their decision we presented frontend developers with experts and software components of the frontend and backend developers with components and experts we computed for the backend.

The second part was a self-assessment where backend developers should state if they could work in the frontend and vice versa. We also wanted to know in which part of the software they think they are well versed.

After the self-evaluation we let every developer choose the experts for their division and for particular software components. For each component at least one expert had to be selected. The developers were then free to name two more developers of whom they think that they also have expertise. This way we were able to determine if the results our framework produces align with the experts the developers have intuitively chosen.



(A) With 80% confidence the King of the Hill for the frontend is Hester Marler.

(B) 100% of participants voted Dania Anstine to be the King of the Hill for the backend.

FIGURE 6.21: The King of the Hill is the most experienced developer. Here as voted by his or her colleagues at Signavio.

Also as none of our results had been presented to the developers until this point we were able to get unbiased insights.

In the last part of the survey we presented the experts we have found using our framework to the developers. We used a ranked list of the top three experts we have found for each component and let the developers express how much they agree with each result by ranking it on a scale from 0 ("I strongly disagree") to 10 ("I strongly agree"). If only less than three contributors could be found using our framework this list was shortened to the respective number of found experts.

6.4.1.1 King of the Hill

Figures 6.21a and 6.21b show the results of the voting for the so-called "King of the Hill" of the frontend and the backend. The King of the Hill is supposed to be the developer who is considered the overall expert for a division. In Figure 6.21 we show all votes which have been cast for developers. The following analysis will focus only on the developers who had the most votes to be either the first, second, or third choice. For example Figure 6.21a would be reduced to only show Hester Marler for the first choice, and Ira Moyer for both the second and third choice. If two developers reached the same score, as Annabelle Prudhomme and Bradford Marston did for the second choice shown in Figure 6.21b, we select the developer who has contributed more revisions to the repository. In this case this means that Annabelle Prudhomme will be selected for the second choice for King of the Hill for the backend.

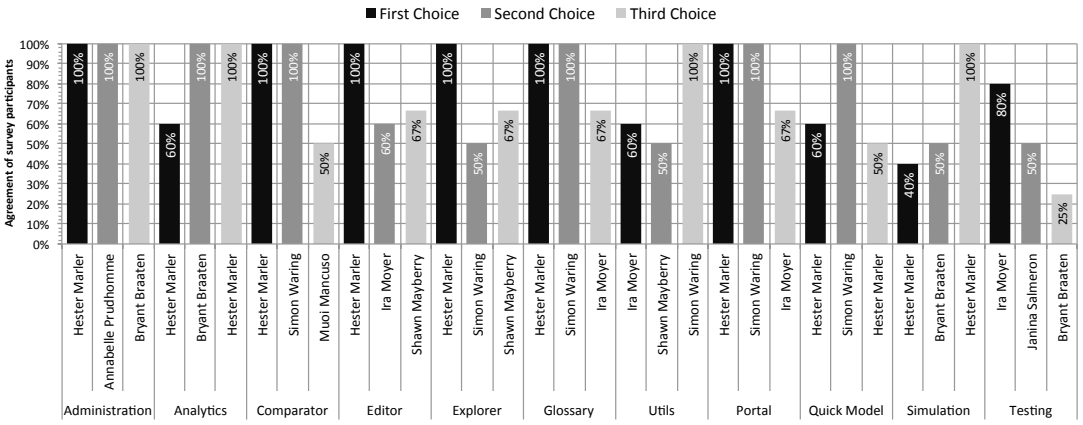


FIGURE 6.22: Experts for the different frontend components as voted by the Signavio staff.

6.4.1.2 Frontend Experts

For the frontend we identified eleven main components as can be seen in Figure 6.22. Each bar represents the part of votes which were cast for the respective developer. We see that for most components the developers agree on the first choice. This means developers have a specific person in mind which they would consult if they had a question. We can also see that there are only two distinct developers who are voted to be the first choice. In nine out of eleven cases this is Hester Marler and in two of eleven cases it is Ira Moyer. This introduces one obvious problem for Hester Marler. He most probably will be interrupted a lot during his day to day work as he is considered the expert for almost every component in the frontend. It also means that if he is out of the office developers would lack their single point of contact if they have any questions.

When it comes to the second and third choice we see more diversity in the answers we got from the developers at Signavio. For both the second and the third choice six distinct developers have been selected. The second choice is dominated by Simon Waring with five out of eleven votes, who is followed by Bryant Braaten with two votes and four other developers with one vote each. Hester Marler again dominates the third choice. We assume that he is chosen as a last stand, meaning that if the developers could not find someone to ask they will fall back to simply asking him.

6.4.1.3 Backend Experts

In the backend we identified six main components as shown in Figure 6.23. We can observe a similar distribution of the developers as we have already seen in the frontend. The first choice is dominated by two developers, namely Dania Anstine who has been chosen four out of six times and Bradford Marston who has been chosen two out of six

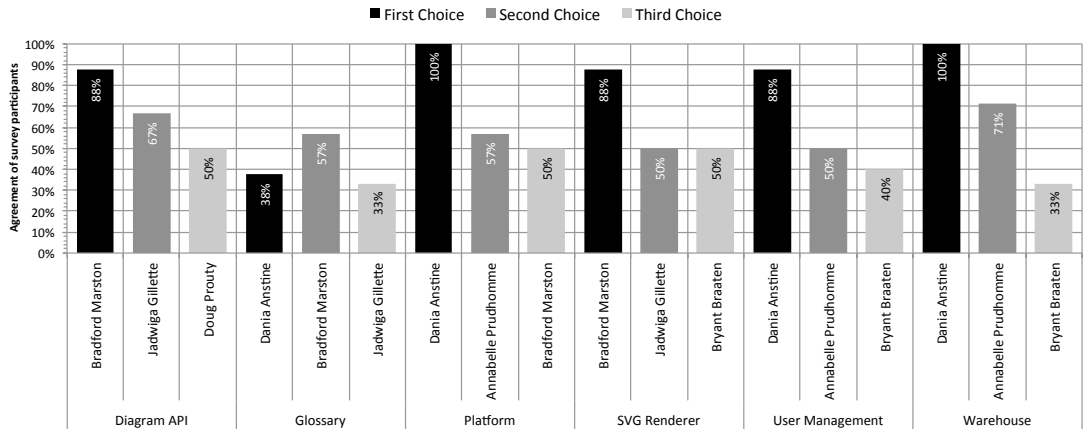


FIGURE 6.23: Experts for the different backend components as voted by the Signavio staff.

times. Annabelle Prudhomme and Bryant Braaten, both with three out of six votes, have the most votes for the second and third choice. From those choices we can also make the assumption that Bryant Braaten is one of the developers who is able to work in both the frontend and the backend because he was chosen as the second choice for components in both divisions.

6.4.2 Evaluation

After we have shown which developers have been voted to be experts by the developers themselves, we want to evaluate the results our framework has produced. In order to do so we presented each developer with the top three experts we have identified for their respective field of work. Our results for the frontend are shown in Table 6.1 and for the backend in Table 6.2. As one can see, we were not able to find three experts for each component. This is caused by the fact that some components are no longer actively developed but only maintained. That way fewer developers are contributing code to the components and therefore loose the ability to become an expert as defined in our framework.

The accuracy of our results will be evaluated in the following manner. If our prediction matches the statement made by the developers we classify it as a full match. Furthermore we also include close misses showing if our prediction missed the top result by one or two places. If our prediction was not included in the statements of the developers it is considered a miss.

Figure 6.24 shows the accuracy of our predictions for the whole repository and also grouped into frontend and backend. In 51.06% of the cases our prediction either matched or only missed the statements made by the developers by one or two places. We managed

		Acceptance			
Component	Experts	0 – 2	3 – 5	6 – 8	9 – 10
King of the Hill	Ira Moyer	0%	0%	60%	40%
	Hester Marler	0%	0%	20%	80%
	Muoi Mancuso	20%	40%	20%	20%
Administration	Jalissa Woods	20%	0%	80%	0%
	Dania Anstine	0%	40%	40%	20%
Analytics	Hester Marler	0%	0%	40%	60%
Comparator	Dania Anstine	40%	20%	20%	20%
Editor	Ira Moyer	0%	0%	60%	40%
	Hester Marler	0%	0%	20%	80%
	Simon Waring	0%	0%	60%	40%
Explorer	Muoi Mancuso	0%	20%	60%	20%
	Hester Marler	0%	0%	20%	80%
	Shawn Mayberry	0%	0%	60%	40%
Glossary	Shawn Mayberry	0%	20%	60%	20%
	Muoi Mancuso	20%	20%	60%	0%
Portal	Hester Marler	0%	0%	0%	100%
	Kirk Moritz	0%	0%	100%	0%
	Ira Moyer	0%	0%	40%	60%
Simulation	Ira Moyer	0%	0%	20%	80%
	Enid Beecham	0%	60%	40%	0%
QuickModel	Kirk Moritz	0%	0%	60%	40%
Testing	Ira Moyer	0%	0%	20%	80%
	Shawn Mayberry	0%	20%	20%	60%
	Hester Marler	0%	20%	20%	60%
Utils	Ira Moyer	0%	0%	20%	80%
	Doug Prouty	0%	0%	40%	60%
	Hester Marler	0%	0%	40%	60%

Table 6.1: The results of our framework for the frontend components showing the experts we found and their acceptance by the Signavio staff.

		Acceptance			
Component	Experts	0 – 2	3 – 5	6 – 8	9 – 10
King of the Hill	Dania Anstine	0%	0%	0%	100%
	Jadwiga Gillette	0%	12.5%	62.5%	25%
	Bradford Marston	0%	0%	37.5%	62.5%
Diagram API	Bradford Marston	0%	0%	0%	100%
	Jadwiga Gillette	0%	12.5%	37.5%	50%
	Dania Anstine	0%	12.5%	62.5%	25%
Glossary	Jadwiga Gillette	0%	0%	12.5%	87.5%
	Bradford Marston	0%	0%	12.5%	87.5%
	Jalissa Woods	0%	37.5%	37.5%	25%
Platform	Dania Anstine	0%	0%	0%	100%
	Bradford Marston	0%	0%	37.5%	62.5%
	Jadwiga Gillette	0%	0%	50%	50%
SVG Renderer	Bradford Marston	0%	0%	0%	100%
	Bryant Braaten	0%	37.5%	25%	37.5%
User Management	Dania Anstine	0%	0%	0%	100%
	Annabelle Prudhomme	0%	0%	25%	75%
	Jadwiga Gillette	0%	12.5%	50%	37.5%
Warehouse	Jadwiga Gillette	0%	0%	75%	25%
	Dania Anstine	0%	0%	25%	75%
	Bradford Marston	0%	0%	62.5%	37.5%

Table 6.2: The results of our framework for the backend components showing the experts we found and their acceptance by the Signavio staff.

to find a perfect match in 28.79% of the cases. If we look at the frontend and backend on their own we can see that the predictions we have made in the backend were more accurate than our predictions concerning the frontend.

After we evaluated the overall accuracy we also looked at the accuracy when we only consider the first choice as this is probably the most interesting one. In Figure 6.25 we show the results of that analysis in the same manner as before. We can see that we managed to exactly match the estimates of the developers in 47.37% of the cases. For the backend this number is even higher with 71.43%. But this analysis only shows if our framework would satisfy the estimations of the developers. If the expert we found did

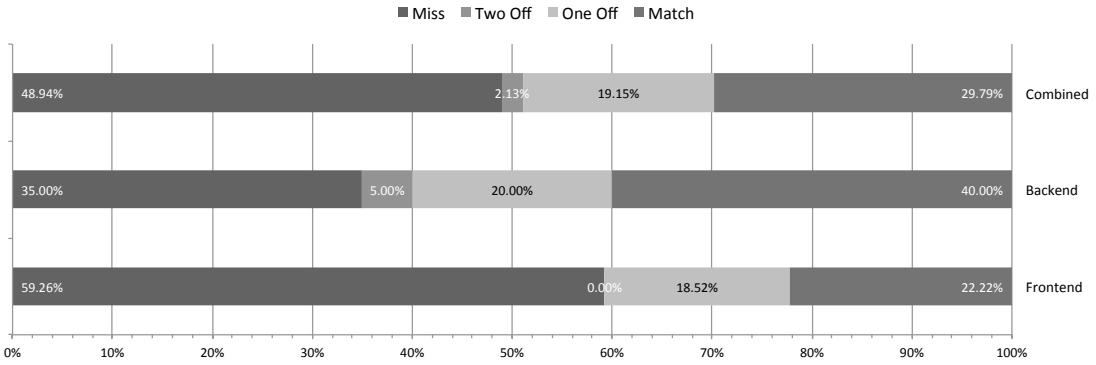


FIGURE 6.24: Accuracy of our findings compared to the statements made by the Signavio staff. The results are grouped into perfect match, one off, two off, and miss.

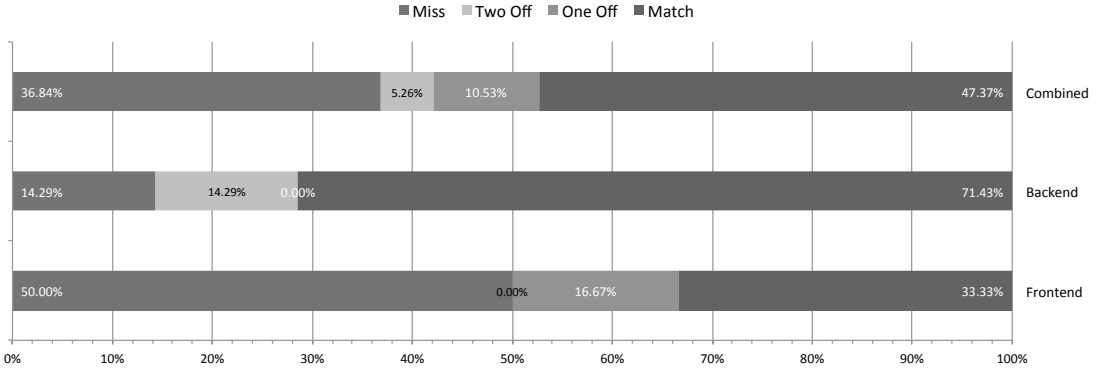


FIGURE 6.25: Accuracy of our findings considering only the first choice compared to the statements made by the Signavio staff. The results are grouped into perfect math, one off, two off, and miss.

not match he or she can still be the right person to ask and the other developers simply did not think of him or her when they filled out the survey.

We argue that once developers have found a person to ask they are likely to ask said person again if they encounter another problem in the future. Therefore the set of people that comes to a developers mind is limited. When we presented the developers with the list of experts we found for certain components, we let the developers vote how much they agree or disagree with our results. We used a scale from 0 to 10, where 0 means "I strongly disagree" and 10 means "I strongly agree". For the following evaluation we grouped the results into four categories.

0 – 2 The developers either strongly disagree with the result or are at least not satisfied. Such a result is of no use.

3 – 5 The result is not absolutely wrong but the developers would also not benefit of it as the found expert does not have enough knowledge in the said field of work.

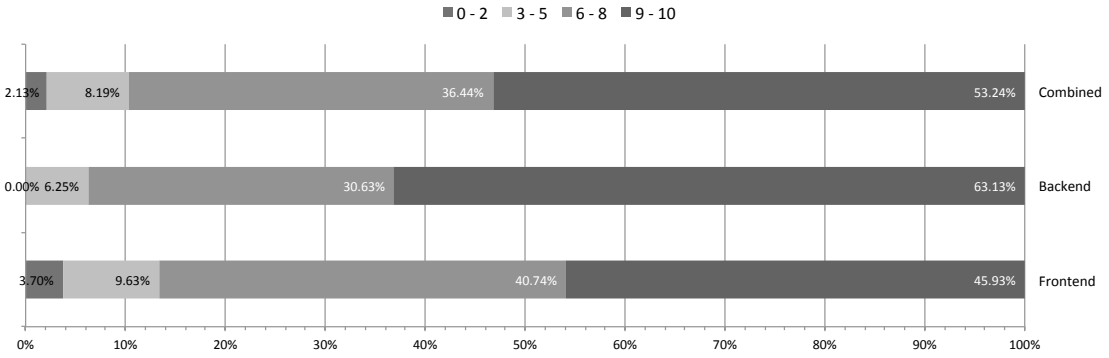


FIGURE 6.26: Acceptance of the computed first, second, and third choice for experts by the Signavio staff.

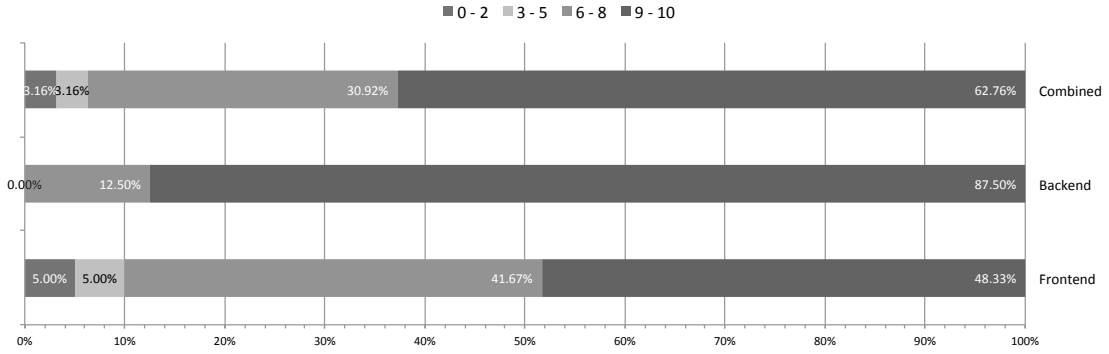


FIGURE 6.27: Acceptance of the computed experts, regarding only the first choice by the Signavio staff.

6 – 8 The expert is generally accepted even though he or she would not have been the first choice of the developers.

9 – 10 The expert we have found is accepted and the result can be considered very useful.

In Figure 6.26 the overall scores for all choices we have made are presented. This means the acceptance for the first, second, and third choice for experts. We can see that 89.68% of our results are accepted by the developers. Developers strongly agree with our results in 53.24% of the cases. Again we managed to have better results in the backend than in the frontend. The full list of results and the respective acceptance can be viewed in Tables 6.1 and 6.2.

The results of this analysis, if we focus only on the acceptance of the first choice we have computed, are shown in Figure 6.27. In 62.76% of the cases the developers fully agree with our first choice. Additional 30.92% agree with the results to an extend that the result would be useful to them. This means that in 93.68% the first choice we presented to developers has been accepted. Results that are not satisfying amount to only 6.32%.

We can therefore conclude that our framework can be used in order to find experts in a software repository. The choices our framework makes are not only based on the amount of revisions a given developer has contributed to a software repository but are refined using static code analysis. Also, selecting only the developers who are currently working with the code influences the result in a positive manner. Future work should further investigate why the results for the backend are consistently better than the results for the frontend. As the backend is written in Java which is a strictly typed language and can therefore be better analysed this might also result in a higher accuracy of the predictions we make. However, without further research regarding this matter it might also be a coincidence.

Chapter 7

Future Work

> *The future is not laid out on a track. It is something that we can decide, and to the extent that we do not violate any known laws of the universe, we can probably make it work the way that we want to.*
>
> – Alan Kay

While working on this theses we often encountered situations where we wanted to extend our research but could not do so because it was out of scope. We want to use this chapter to present the most relevant ideas that came up along the way and which we think should be covered by future work.

7.1 Code Metrics

More code metrics could be used in order to refine the results of our framework. In particular the structure metrics should be reevaluated and measures such as the instability index [44] should be included in the analysis. The instability index gives insights about how reusable a package is by examining the abstractness and instability of classes. A class should have a level of abstractness so that it can be reused and does not have to be rewritten for each use case. At the same time it must be concise enough so that it can perform actual work and is not completely useless.

Another metric which should be examined in more detail is the information flow metric introduced by Henry et. al, which correlates the efferent and afferent coupling of classes with their respective amount of lines of code [12, 44–46]. Of course, as stated in Section 6.3, we first must define a more valuable mapping for the values of C_e and C_a into the Squale model in order for such measures to add value to the overall result.

Even though we found a method to aggregate software metrics efficiently, we think that we could gain even more insights if we set the level of granularity at which we store metrics to functions instead of classes or files. We would be able to not only assess how the complexity of files changes, but also which methods are changed most of the time. We argue that this would make it feasible to determine in which functions errors are most likely to originate and which methods therefore need extra testing. Also, more focus should be put on the files whose metric values leave a good range and decrease to critical ranges.

In order to incorporate the change rate of certain files into the metrics we propose that code churn measures as shown by Nagappan et. al should be included in the framework. This way stronger restrictions regarding complexity or structural metrics can be applied to those parts of the code that show the highest change frequency [17]. The code churn could further be used in order to assign a weighting to the changes a developer has caused. As stated in Section 6.4 we currently use the ratio of increases to decreases of the metrics to distinguish between developers which created a similar number of revisions. This penalises developers who improve large amounts of the code, as they get the same score as developers who only change very small parts. We therefore propose that future work will use the amount of committed lines of code to further weigh the quality improvements and degradations.

If we shift the focus away from software metrics in the classical sense, we could include the automatic detection of code duplication. Duplicates in the source code do not necessarily increase the complexity but in any case make it harder to maintain. A duplicated component easily tricks a developer into believing that he or she has fixed an error throughout the system, but in fact the error was only fixed in the copy of the code which was found by the developer.

7.2 Incorporating the Development Method

As we not only record the revisions which have been created inside a software repository but also save their dates, we might be able to detect releases of a software as proposed by Alali et. al [47]. We think it would be interesting to see if the software metrics behave differently during the days leading up to a release and those following it. When we further combine this information with the code churn we record for the different files and packages we might be able to determine troublesome parts inside the software. Keogh et. al propose algorithms to efficiently perform pattern matching on time series data [48, 49], which we could employ in order to recognise patterns that lead to releases with a higher bug count. We argue that unstable releases are followed by a high amount of

FIGURE 7.1: Punchcard heat map showing during which times of the day the most code is contributed to a repository.

code churn which represents the effort of developers to quickly deliver hot fixes for the most problematic errors.

The work of Eyolfson et. al points out that the quality of code also highly correlates with the time during the day when the code was written [40]. Tired developers are more likely to introduce errors and therefore complex code that is committed during the night is likely to be prone to errors. Our framework partly supports such analysis as shown in Figure 7.1, yet it was not possible for us to further investigate this matter in this thesis.

7.2.1 Differentiation by Programming Language

During our analysis we found that a distinction between different programming languages is sensible. As different programming languages also enforce different programming paradigms, the assumption that there might be one scale which can be used in order to universally assess the quality of code is wrong. The main difference is the amount of code that needs to be written in order to produce a working program. When we evaluated the approach to find the main contributors to a software repository in Section 6.1.4, we found that it would be beneficial to distinguish between the different kinds of programmers that can be found inside an organisation. In our case this distinction would be made between frontend and backend developers. By doing so, we would no longer base this measure on the amount of commits a developer creates inside a repository, as this number highly depends on the commit behaviour of the individual developer (as shown in Section 6.1.1). It is rather based on the actual amount of code the developer has contributed. We think this would be a more reliable source of information if we want to find the main contributors. It also would be interesting to see if and to what extent the main contributors we find with this approach would differ from the ones we have currently determined.

7.3 Knowledge Management

In Section 6.4 we have shown our approach to determine experts inside a software project. For each component we tried to discover the top three developers who would be able to help other developers when they face a problem inside the source code. We also noticed that we could not in any case compile a list of three experts. While this is not very helpful for the developer who is seeking help it reveals parts of the software which might need special attention in the sense of documentation. Depending on how long there has not been any activity we might be able to find either dead code which is no longer used or code which should be documented while there are still developers in the organisation who know something about it. Otherwise developers with knowledge concerning those parts of the software could vanish completely from the organisation which could lead to major efforts when defects are detected in the respective code.

We might also be able to detect parts inside an application that seem to be very prone to errors using our framework. Parts of the software with a high code churn rate and low metric values are especially critical and should therefore be either refactored or well documented.

De Souza et. al show how the relationship between developers and the files they author changes over time [50]. This information might be further leveraged in order to determine teacher-apprentice-relationships automatically. If we see that a developer's activity slowly decreases, while a new developer contributes more and more changes to the code, this might mean that one developer is the teacher or mentor of another one. We might even be able to detect whether developers interact with each other if we observe alternating commits of two developers on the same code. Clearly, this is a bold statement to make and it should be further evaluated. But if we were able to draw such conclusions we could help to appoint a successor if a developer leaves the organisation.

Chapter 8

Conclusion

In this thesis we have shown how our framework can be used in order to gain insights from the source code of software projects. Even though Yashamita et. al claim that software metrics alone cannot be used to assess source code [14], we have shown that they can, for instance, be used in order to automatically find experts inside a group of developers. To gain valuable results, we relied on work by German et. al [5] and others to select metrics that produce the most insightful results. We validated our approach with a case study at a successful German startup. The developers at Signavio have approved our results via a survey. Though we do not claim that our framework will produce similarly accurate results for every organisation, we nevertheless believe that it certainly will add value and improve various parts of the software development process. The results of our expert search provide developers with someone who is most likely suited to offer them help and guidance if they encounter a problem. We argue that we can relieve pressure from those developers who act as the single point of contact for questions inside an organisation, as the experts we determine might not always be the obvious choice. This way we also distribute knowledge to a broader group of developers. Additionally, highlighting an uneven distribution of experts is likely to encourage organisations to push collective code ownership which has been proven to increase the overall performance of software development teams [43].

In order to reduce workload mismatches caused by a disadvantageous distribution of workforce, we have shown how developers can be automatically categorised based on their primary field of work. We also determine developers who are able to help out in other divisions if needed. As our data reflects the current staffing situation of an organisation, decisions based on our results are likely to improve the overall productivity.

The Analyzr framework which was developed to perform the measurements for this thesis is being made available freely under the MIT license for interested parties. As the system

was designed as a web application and is currently compatible with both SVN and Git repositories, it can be easily integrated into an existing SCM landscape. Support for other versioning systems can be added by implementing a small interface as described in Section 3.2.1. A basic user management system also allows to restrict the access to certain developers.

Our framework and the results presented in this thesis encourage developers to write code which is less complex and therefore easier to maintain. Using the Analyzr framework, each developer can view his or her own statistics and see the progress he or she is making. The included mechanism to anonymise the results can further be used in order to maintain privacy and minimise the pressure some developers might feel when such a framework is introduced to the development process. Apart from that, it can help management to establish an atmosphere where developers are encouraged to discuss their code and learn from each other.

In addition to the aspect of feasibility, which is demonstrated by our implementation, we have shown that our approach is viable as it is likely to help organisations to improve their development process while increasing the maintainability of their code base. On the other hand it is also desirable for developers as we are able to help them during their day to day work, unburdening developers who are currently the single point of contact and spreading knowledge throughout the organisation.

Bibliography

[1] Horst Zuse. 1 history of software measurement. 1998.

[2] Alexandru G Bardas. Static code analysis. *Journal of Information Systems & Operations Management*, 4(2):99–107, 1994.

[3] William Thomson Baron Kelvin. *Popular lectures and addresses*, volume 3. Macmillan and Company, 1891.

[4] Karl E Wiegers. Software process improvement: ten traps to avoid. *Software Development*, 4(5), 1996.

[5] Andy German. Software static code analysis lessons learned. *Crosstalk*, 16(11), 2003.

[6] Thomas J McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976.

[7] Arthur H Watson, Thomas J McCabe, and Dolores R Wallace. Structured testing: A testing methodology using the cyclomatic complexity metric. *NIST special Publication*, 500(235):1–114, 1996.

[8] MH Halstead. Potential impacts of software science on software life cycle management. 1977.

[9] Cem Kaner and Walter P Bond. Software engineering metrics: What do they measure and how do we know? *methodology*, 8:6, 2004.

[10] Victor R Basili, Lionel C. Briand, and Walcélio L Melo. A validation of object-oriented design metrics as quality indicators. *Software Engineering, IEEE Transactions on*, 22(10):751–761, 1996.

[11] Barry W Boehm. Software engineering economics. *Software Engineering, IEEE Transactions on*, (1):4–21, 1984.

[12] Sallie Henry and Dennis Kafura. Software structure metrics based on information flow. *Software Engineering, IEEE Transactions on*, (5):510–518, 1981.

[13] Mika Mantyla, Jari Vanhanen, and Casper Lassenius. A taxonomy and an initial empirical study of bad smells in code. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 381–384. IEEE, 2003.

[14] Aiko Yamashita and Leon Moonen. Do code smells reflect important maintainability aspects? In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 306–315. IEEE, 2012.

[15] Eva Van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, pages 97–106. IEEE, 2002.

[16] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, 1994.

[17] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 284–292. IEEE, 2005.

[18] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, pages 452–461. ACM, 2006.

[19] M Clark, B Salesky, C Urmson, and D Brenneman. Measuring software complexity to target risky modules in autonomous vehicle systems. In *Proceedings of the AUVSI North America Conference*, 2008.

[20] R Harrison, S Counsell, and R Nithi. An overview of object-oriented design metrics. In *Software Technology and Engineering Practice, 1997. Proceedings., Eighth IEEE International Workshop on [incorporating Computer Aided Software Engineering]*, pages 230–235. IEEE, 1997.

[21] Mehwish Riaz, Emilia Mendes, and Ewan Tempero. A systematic review of software maintainability prediction and metrics. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 367–377. IEEE Computer Society, 2009.

[22] Doug Hoffman. The darker side of metrics. In *Pacific Northwest Software Quality Conference, Portland, Oregon*, 2000.

[23] ISO ISO. Iec 25000 software and system engineering–software product quality requirements and evaluation (square)–guide to square. *International Organization for Standarization*, 2005.

[24] Robert Baggen, José Pedro Correia, Katrin Schill, and Joost Visser. Standardized code quality benchmarking for improving software maintainability. *Software Quality Journal*, 20(2):287–307, 2012.

[25] José Pedro Correia, Yiannis Kanellopoulos, and Joost Visser. A survey-based study of the mapping of system properties to iso/iec 9126 maintainability characteristics. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 61–70. IEEE, 2009.

[26] Todd L Graves, Alan F Karr, James S Marron, and Harvey Siy. Predicting fault incidence using software change history. *Software Engineering, IEEE Transactions on*, 26(7):653–661, 2000.

[27] Vincent Yun Shen, Samuel D. Conte, and Hubert E. Dunsmore. Software science revisited: A critical analysis of the theory and its empirical support. *Software Engineering, IEEE Transactions on*, (2):155–165, 1983.

[28] Alexander Serebrenik. 2is55 software evolution. 2011.

[29] Douglas Crockford. *JavaScript: the good parts*. O'Reilly Media, Inc., 2008.

[30] Thomas Zimmermann and Peter Weibgerber. Preprocessing cvs data for fine-grained analysis. 2004.

[31] Bill Karwin. Sql antipatterns. *Avoiding the pitfalls of database programming. The pragmatic bookshelf*, 2010.

[32] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[33] Bogdan Vasilescu, Alexander Serebrenik, and Mark van den Brand. By no means: A study on aggregating software metrics. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Metrics*, pages 23–26. ACM, 2011.

[34] Bogdan Vasilescu, Alexander Serebrenik, and Mark van den Brand. You can't control the unfamiliar: A study on the relations between aggregation techniques for software metrics. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 313–322. IEEE, 2011.

[35] Karine Mordal, Nicolas Anquetil, Jannik Laval, Alexander Serebrenik, Bogdan Vasilescu, and Stéphane Ducasse. Software quality metrics aggregation in industry. *Journal of Software: Evolution and Process*, 2012.

[36] F. Balmas, F. Bellingrad, F. Denier, S. Ducasse, B. Franchet, J. Laval, K. Mordal-Manet, and P. Vaillergues. The squale quality model. modèle enrichi d'agrégation des pratiques pour java et c++. Technical report, INRIA, 2010.

[37] Gero Decker, Hagen Overdick, and Mathias Weske. Oryx–an open modeling platform for the bpm community. In *Business process management*, pages 382–385. Springer, 2008.

[38] Ken Schwaber. Scrum development process. In *Business Object Design and Implementation*, pages 117–134. Springer, 1997.

[39] John Resig. jquery: The write less, do more, javascript library. *Taken from http://jquery.com/, Accessed on 21.02.2014*, 2014.

[40] Jon Eyolfson, Lin Tan, and Patrick Lam. Do time of day and developer experience affect commit bugginess? In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 153–162. ACM, 2011.

[41] David Schuler and Thomas Zimmermann. Mining usage expertise from version archives. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 121–124. ACM, 2008.

[42] Thomas D LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, pages 492–501. ACM, 2006.

[43] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code!: examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 4–14. ACM, 2011.

[44] Edwin Hautus. Improving java software through package structure analysis. In *The 6th IASTED International Conference Software Engineering and Applications*, 2002.

[45] Alexander Serebrenik, Serguei Roubtsov, and Mark van den Brand. D n-based architecture assessment of java open source software systems. In *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*, pages 198–207. IEEE, 2009.

[46] Robert Martin. Oo design quality metrics. *An analysis of dependencies*, 1994.

[47] A. Alali, H. Kagdi, and J.I. Maletic. What's a typical commit? a characterization of open source software repositories. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 182–191, 2008. doi: 10.1109/ICPC.2008.24.

[48] Eamonn Keogh. A fast and robust method for pattern matching in time series databases. In *proceedings of 9th International Conference on Tools with Artificial Intelligence (TAI'97)*. Citeseer, 1997.

[49] Eamonn J Keogh and Padhraic Smyth. A probabilistic approach to fast pattern matching in time series databases. In *KDD*, volume 1997, pages 24–30, 1997.

[50] Cleidson De Souza, Jon Froehlich, and Paul Dourish. Seeking the source: software source code as a social and technical artifact. In *Proceedings of the 2005 international ACM SIGGROUP conference on Supporting group work*, pages 197–206. ACM, 2005.

List of Figures

List of Tables

The End.