A real-world example of Transfer Pricing in Inner Source

MASTER THESIS

Stefan Buchner

Submitted on 19 May 2021

Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik
Professur für Open-Source-Software

Supervisor:
Prof. Dr. Dirk Riehle, M.B.A.

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 19 May 2021

License

Erlangen, 19 May 2021

Abstract

Inner source is understood as the application of the open source paradigm within organizations. Inner source enables developers to contribute to software modules not only within their own organization, but also to those belonging to other organizations within the same company. This means from a developers point of view an increase of flexibility. However, for management, accounting and taxation transferring work between independent business entities can lead to significant challenges, as tax boundaries are crossed and therefore, transfer prices must be calculated. The main challenge here is, that calculating transfer prices for mixed software development is hard, as no exact data about work time is available.

To assign work time to single projects, a list of code contributions (commits) were analyzed to find logical structures. Based on this analysis, a work time calculation concept was developed. In addition to the concept development, a prototype for evaluation was developed. This implementation also includes an example transfer price calculation for taxation purposes, conducted by the Cost-Plus method.

Contents

Acronyms

OECD Organisation for Economic Co-operation and Development

HGB Handelsgesetzbuch

EStR Einkommensteuer-Richtlinien

IFRS International Financial Reporting Standards

LOC Lines of Code

REST Representational State Transfer

CSV comma-separated values

1 Introduction

For various business reasons it is important to measure costs and time spend on products. Most commonly to track costs (controlling), for decision making (strategic management), and to track product development progress itself. Doing this for physical goods can be easy. For intellectual property and services, measuring costs and work time is harder.
As long as work time for services and intellectual property can be clearly separated (one department, one product/service), there are less problems in calculating costs. This is also true for software development. However, problems arise if software development crosses organizational boundaries. Even though calculating costs for entire organizations seems relatively easy in the first place, doing this for collaborative development is not. It makes the life of programmers easier, if they can work on project useful to their goals, but not limited to their organization. Therefore, it would increase efficiency if developers could work without minding these organizational boundaries from a managerial point of view.

Development work crossing organizational boundaries (especially with inner source development) affects for example taxation (In which country was the development done?), controlling (What where developer working on? Which products produce how many costs?) and management (How can i steer the people so that development is more efficient?)

The main problem is that due to the high frequency of work contributions in software development, there is no direct way to differentiate how the work effort of a department splits between certain products.

This thesis aims to solve the problem of assigning work time and splitting costs of collaborative software development based on the list of code contributions (commits). Moreover, it does not only present an algorithm to calculate work time, but also shows a prototype implementation. The goal is to determine work time for costs calculation for the use case of transfer pricing in inner source software development, especially for taxation purposes using the cost plus method.

The thesis is structured as followed: First of all, the foundations and relevant literature are explained (Chapter 2). In this chapter, the basics of transfer pricing,

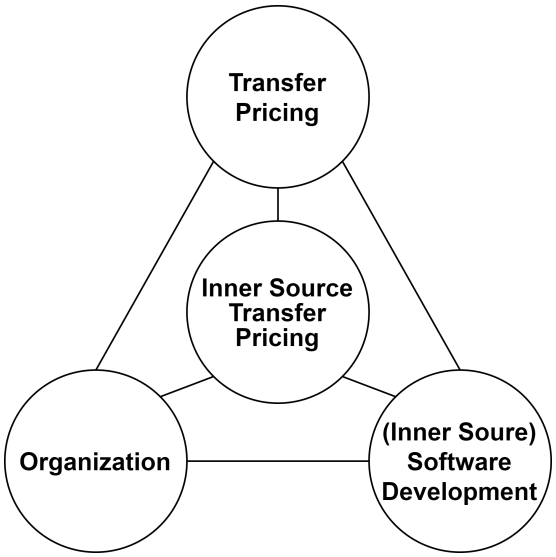
organization forms and the inner source paradigm are shown individually and finally brought together to motivate the need for Inner Source Transfer Pricing.
After the motivation, the work calculation concept (an algorithm) and its development process is presented (Chapter 3).
The implementation of the algorithm is then shown with a prototype which calculates the work time and an example transfer price using the cost plus method (Chapter 4).
Next, some ideas and concept how an evaluation can look like are presented (Chapter 5).
Finally, the thesis is summarized and its limitations and further research potentials are discussed (Chapter 6).

2 Fundamentals

Inner Source Transfer Pricing as an interdisciplinary topic is based on and affects different sciences and subjects. In this chapter, the fundamentals of each subject are explained individually. Afterwards the topics are connected to show the problems currently existing for Inner Source Transfer Pricing.

First, Inner Source Transfer Pricing is a topic with economic impact and background. It contributes to make inner source software development economical more viable, therefore some basic economic principles need to be explained (Chapter 2.1). Secondly, inner source is also a way to organize programming teams, departments and the way information flows between those (See Chapter 2.2). Afterwards the core elements of the inner source programming paradigm are explained, how it originates and differs from open source development and what the differences to traditional programming paradigm commonly used in companies are (Chapter 2.3). Finally, the dependencies between those basics are explained (Chapter 2.4). This chapter additionally motivates the need of the concepts developed in this thesis from different perspectives. An overview of the three basic topics can be seen in Figure 2.1.

Figure 2.1: Dependencies between Inner Source Transfer Pricing topics



2.1 Economical

Inner source software development can affect and improve various economic topics like taxation, controlling, strategic management and personnel management. In this work, the focus is on taxation, even though some methods used here are close to those used in controlling. Future research must show, if the result of this work is also suitable for a use in controlling.

At first, it is important to know, what transfer prices are and what approved methods exists to calculate those. Afterwards the importance of transfer pricing for companies is showed. At last, the full absorption costing as basis for the Cost Plus taxation method is explained.

2.1.1 Transfer pricing principle

First, the term transfer pricing has to be explained generally without being too specific about inner source. The OECD (2017b, pp. 33–34) described the need for transfer pricing as followed: In an ordinary market, two independent market participants determine the price of a product or (intellectual) property according to the price of the overall market and what other participants are offering. For a transfer between two connected companies (e.g. both are part of the same group/holding), not always a market exists. This makes the price calculation more complicated.

However, for the government and cooperation the height of the price is important, as it influences the tax earnings and the economic situation of the participating

companies (OECD, 2017b, p. 34). Prangenberg et al. (2011) explicitly added that the calculation of transfer prices does not only affect goods, but also services. As programming is a mainly a service with (as to expect) a lower amount of tangible assets, it is harder to assign a plausible value to a transaction as when goods are produced.

2.1.2 Transfer pricing relevance for companies

A dedicated look on how transfer pricing is influencing companies helps to understand the need and use for the concept developed in this thesis. As the OECD (2017b, p. 34) stated, transfer pricing influences the cash-flow of a group, the shareholder value and overall profitability.

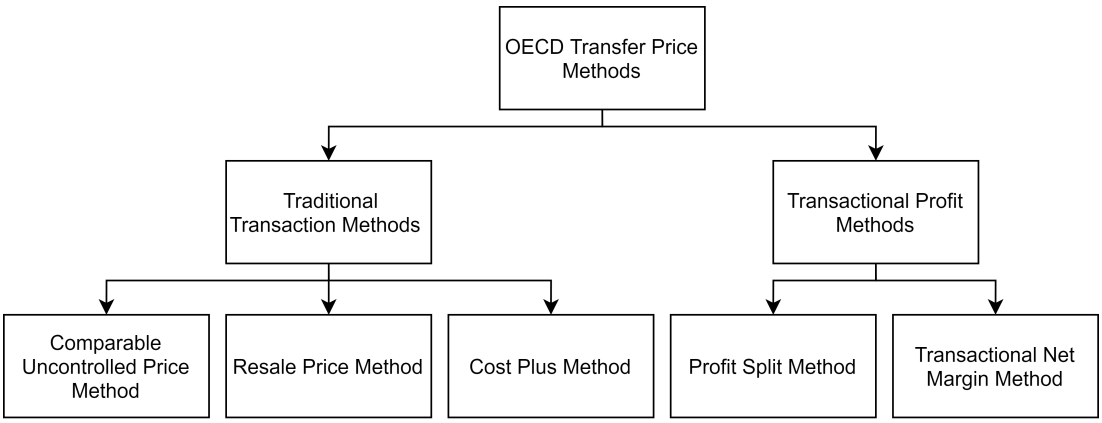
Prangenberg et al. (2011, pp. 19–28) are mentioning three categories in which companies profit from transfer pricing: The steering function, income distribution function and control function. Even though not all functions are directly correlated with Inner Source Transfer Pricing, the thesis helps to improve these areas. As an example, Inner Source Transfer Pricing helps to split the profit between connected companies. Consequently, tax risks can be reduced, and a target/actual comparison can be simplified.

Hanken et al. (2017, pp. 23–28) structured the way companies profit from transfer pricing similarly. However, additionally they differentiate between a controlling and a taxation view. This differentiation is not only important for the possible future use-cases of this thesis, but also to understand the need to quantify inner source development.
The controlling view on transfer pricing is for the internal use. Prices calculated there are not for outside use, but for strategic management support.
The tax view on the other hand calculates transfer prices for taxation use and how the profit is split between connected companies. Hanken et al. (2017) are giving a deep insight into possible ways to calculate tax transfer prices (Chapter B) and those for internal use (Chapter C). Additionally, they show the correlation between those methods and who they can be combined (pp. 569-586).
Even though the concept and software in this work was mainly developed for taxation use, the re-use for other (e.g. controlling) purposes was always kept in mind during development. Chapter 2.4 will go more into detail, what the dependencies between inner source and transfer pricing are and how it can be used for controlling and taxation.

2.1.3 Taxation methods

As said in the previous section, the transfer price methods used for controlling and taxation might overlap at some points, but this work is focusing on taxation methods to calculate transfer prices. In this chapter basic methods are shown,

Figure 2.2: Overview of transfer price methods, adopted by (Schwerdt, 2016, p. 166)



which can also be seen in Figure 2.2. The five most common methods for transfer pricing are assigned to two major categories.

The first set of methods are the traditional transaction methods which based on comparing the transaction to one which unconnected companies would have done on the market (Schwerdt, 2016, pp. 165–166).
The second set of methods are called transactional profit methods. They are looking at the profit a transaction brings and examines if it was influenced by to the fact, that it is a controlled environment (OECD, 2017b, p. 117).

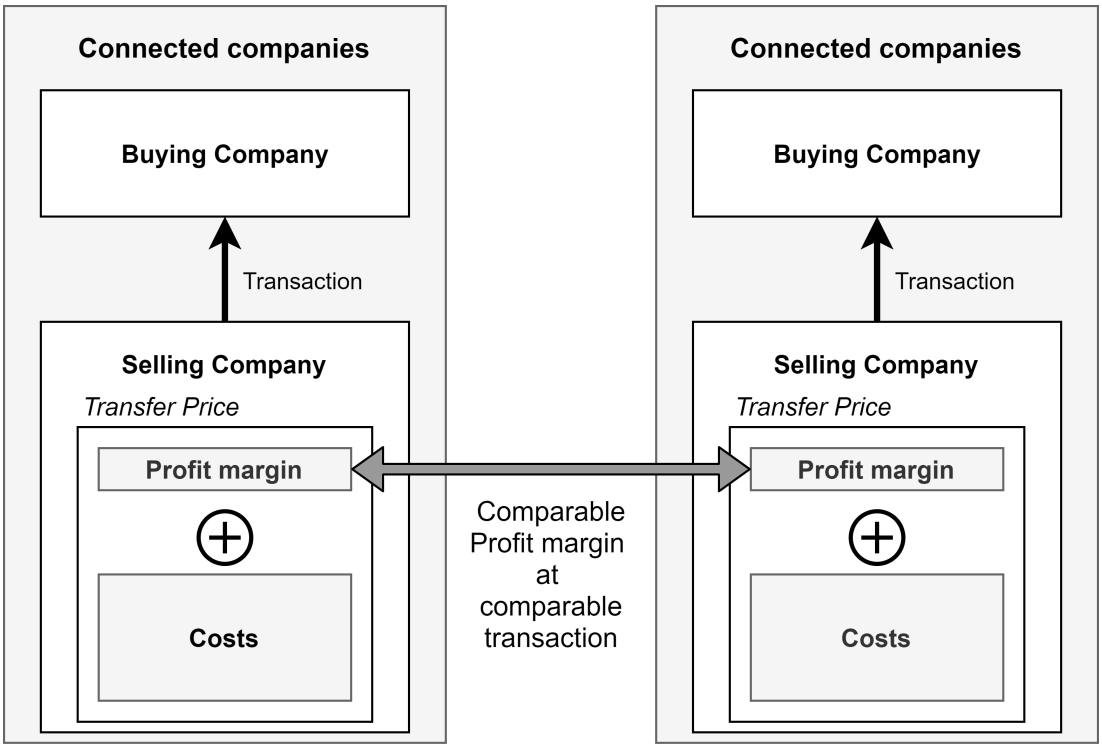
Comparable Uncontrolled Price Method:
OECD (2017b, pp. 101–103) describe the Comparable Uncontrolled Price Method as a method, where the transfer price is set equally to a transaction comparable to one we are looking at. The transaction to which the current one is compared must not be under own control, therefore not be a connected company.

Important to know is also, that the circumstances under which the comparing transactions are happening must be similar, even small differences can influence the transfer price (OECD, 2017b, pp. 101–105).
Simply said, the method directly compares the transfer price.

Resale Price Method:
The Resale Price Method has a close relation to the Comparable Uncontrolled Price Method. In contrast to the latter, the former does not compare the transfer price directly, but the resale price of the products on the market, as (Schwerdt, 2016, p. 176) presented. They emphasize, that the resale price is reduced by a margin which helps the selling company of the observed transaction to cover its costs and make an own profit.

According to the OECD (2017b, pp. 105–106) the margin can be determined by

Figure 2.3: Cost Plus procedure



comparable internal sales or also to margins of a comparable external sale.

Cost Plus Method:
The next method is the Cost Plus Method, which does not look at the resale price or the transfer price directly, but to the costs associated in the production or service delivery. OECD (2017b, p. 113) states, that this method might be supported by other methods. Therefore, the Cost Plus Method can widely be used to support other methods.

The basic principle behind Cost Plus is simple(See Figure 2.3). A cost basis must be calculated, before a profit margin is added on top. Schwerdt (2016, pp. 182–183) made clear, that the profit margin must be suitable to risk, function, and the market conditions of the observed transaction. Moreover, they stated, that the margin must be comparable to other margins in the market (external) or within the same company(internal).

Prangenberg et al. (2011, pp. 55–58) explained that any acknowledged cost calculation method can be applied. As the Cost Plus Method is a traditional transaction method which base on comparison to external factors, the applied cost calculation method must also able to be used for calculation towards externals or at least be based on economic principles.
One commonly used method for cost calculation is the full absorption costing,

but also partial cost accounting might be valid in some use-cases. Moreover, Prangenberg et al. (2011, pp. 56–57) emphasized, that there is also the choice what type of costs to use (direct vs indirect) or which point of time is used to calculate the price (actual/target/plan).

Profit Split Method:
The Profit Split Method is the first transactional profit methods to be looked at. This method controls how the profit is split in a controlled environment, as OECD (2017b, p. 144) described. When using this method, the profit of a transaction between two controlled companies must be comparable to profit a transaction between two independent enterprises would have had.

Transactional Net Margin Method:
The last method to look at is the Transactional Net Margin Method. As this method belongs to the category of the transactional profit method, it takes a deeper look into the profit. In comparison to the Profit Split Method, not the height of the profit split between two companies is important, but the height of the net margin itself.
The net margin is compared to an appropriate base (like costs, sales, assets)(OECD, 2017b, p. 117). Like the methods explained before, the values used must be comparable. In this case, the base must be comparable to external and internal values. OECD (2017b, p. 117) explicitly mentioned that this method might not be appropriate, if a party contributes a unique value to the transaction, as those are not easy comparable.

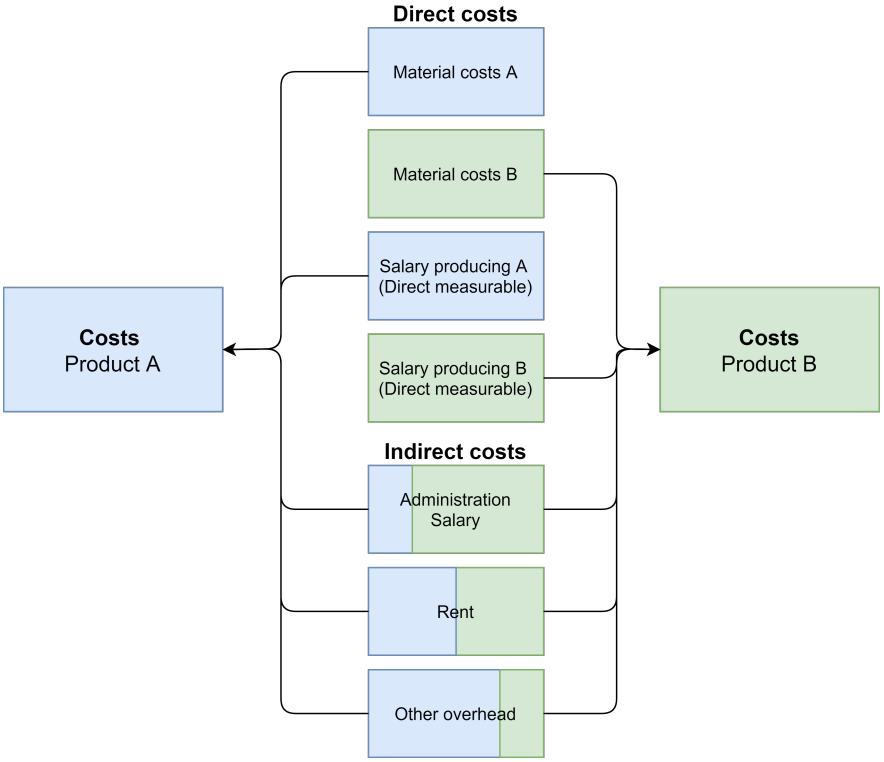
To sum up the different methods looked so war, it can be seen, that all methods are basing on comparing certain values to the market or to equal internal transaction. All methods have in common, that they are comparing different aspects or steps in a value chain. That can be the production costs and margin (Cost-Plus), the profit split of the transaction, the transfer price itself, the resale price, or any other base value the margin can be compared to.

For future research, it might be reviewed, whether the concept and software developed here might be suitable for all methods introduced. However, this work is focusing on the use of the Cost-Plus Method, as the nature of the data given and task to be solved fit best to this method.
A more detailed explanation will be given in Chapter 2.4.2 as first the nature of the inner source paradigm and it specialties needed to be explained (Chapter 2.2 and 2.3) to fully understand the reasons for using Cost Plus.

2.1.4 Full absorption costing

As the Cost-Plus Method is used in this work, a detailed look at how full absorption costing is working is necessary.

Figure 2.4: Full absorption costing procedure



First, it is important to understand how the basic process is working. In the following, full absorption costing will be explained as Hanken et al. (2017, pp. 461–473) described it. The previous chapter already indicated the principle of full absorption costing: It is a way to include all costs which occur during a production process or service delivery.
The term itself does not specify one fix procedure. The way to calculate and decision which costs to include is depending on the accounting scheme used and country the company is located it. In Germany for example the standards of HGB, EStR and IFRS might be applied, depending on the use-case, whereas the EStR is the tax regulation that might be applied to Inner Source Transfer Pricing. However, the decision which standard is chosen for inner source will not be a part of this work, as this decision might come with many considerations to be made in practice. Depending on the accounting standard chosen, some cost can, must or mustn't be included in the calculation.

For the cost calculation itself, two type of costs must be differentiated, which have a different connection to the good being produced. On the one hand there are direct costs. Those are (as the name says) in height directly correlated to the good being produced. Classical examples are the materials for production. On the other hand, there are indirect costs. Those do exist in the company

independently from the amount of goods being produced. Practical examples might be overhead for management, production or marketing. An example how the split between direct and indirect costs look like can be seen in Figure 2.4.

To calculate the costs a good or service produces, all direct costs must be added together. All the indirect costs must be split down to an appropriate ratio which reflect the effort spend with producing the good or conducting the service. After the cost are split, they are also added to the direct costs.

However, if this scheme is applied to software engineering, some problems are occurring. As software engineering is more a service and less production with hardware associated to it, it is harder to differentiate between direct and indirect costs. Developers whose are dedicated to one specific project might be easy and directly assigned to the product designed. Nevertheless, those developers who are working on different products or bringing support work are indirect costs as their work time is split up.

The concept provided in this thesis helps to tackle the cost split problem for software engineering, especially occurring in inner source development. Chapter x shows gives a more detailed look from the inner source paradigm point of view.

2.2 Organizational

Beside transfer pricing and its calculation, the way an organization can be structured is the second important business fundamental needed to know. On the one hand, two traditional organization types will be explained: Functional and divisional organization. On the other hand, the platform organization is described, what advantages it has for software engineering (especially inner source) and problems of the traditional forms it solves.

The way a company structures its organization is important, as this structure also shows in the product the entity it is designing (Conway's law). Therefore, an organization not suitable for designing software, but for physical products leads to inefficiency, which will be explained in this chapter.
Additionally, the type of organization used for software is also connected to and influences transfer pricing and inner source. The explicit connection to inner source will be considered in chapter 2.4.1.

2.2.1 Traditional organization types

Two commonly used ways to internal structure and organization are the functional and divisional organization. Even though in practice more organizational principles exist, this work focusses just on these two as those show practical the

problems of an inefficient structure for software engineering (especially in case of inner source and platform development).

Weber et al. (2014, p. 115) explained the functional organization as one, where each department is responsible for a specific type of business task. This can be procurement, production, or sales. In this way or organizing a company has one department for each type of task to be fulfilled. Consequently, this leads to a high specialization for each task, but also to a higher separation between departments. As Weber et al. (2014, p. 115) described, this kind of organization is structure is mostly fitting for companies which produce a single kind of good requiring similar knowledge (e. g. market needs, production etc.).

The contrast to a functional organization is the divisional organization, which Weber et al. (2014, p. 116) specified as an organization, where a single company is split into several smaller divisions. Those divisions are again structured like a functional organization. Reason to split a company into divisions is that it produced different kind of products which need special knowledge in each area. Therefore (as an example), the procurement department can more easily see the market need for its products. On the contrary this leads to redundancies as each department needs its own divisions.

In the transfer pricing description in Chapters 2.1.1 and 2.1.2 where described, that calculating transfer prices is related to transactions made between two connected companies. Taking the definition for functional or divisional organization into consideration, it can be seen, that the two connected companies might be different functions or divisions of a business group.

Applying the functional or divisional organization structures to software development means, that for example development teams for different products are assigned to extra organizations and therefore treated individually. Considering Conway's law it can make sense to use functional or divisional organization, if it is reasonable, that the product developed inherits the structure given by the organization.

2.2.2 Platform and inner source organization

As said in the previous section, organizing software development with a traditional functional or divisional organization might make sense, if the product more or less inherits the structure of the organization.
However, as software is getting more complex over the years, functional organization brings problems for programmers with it.

Fluri and Deck (2018, pp. 259–261) described one of the main problems functional organization is bringing with it in modern software development. Splitting development strict into functions makes problem-solving and communication between

those departments harder as each team sees its own responsibilities mainly in its own tasks. Consequently, higher coordination costs and development times are arising. Fluri and Deck (2018, pp. 259–261) mainly showed that problem for the split between IT Development and Operations and proposed DevOps as one possible solution to this problem.

However, these problems do not only occur between operation and development, but also within the development. Fuller (2019) confirmed this and even considered functional organization harmful and counter-productive for software development. He also emphasized, that this is especially for cross-platform products the case.

Considering these problems, a new way to organize teams is needed to fit the special need of software developers. Inner source development is (from an organizational point of view) a paradigm specially designed to fit products which are based on a platform and might be inefficient to develop being functional structured.
What the inner source programming paradigm exactly means from a programmers point of view will be discussed in the next chapter. From an organizational view it means, that product development is opened up in a way, that a single developer is no longer responsible for (a single part of) a product but can work on many different software (-parts). This can be helpful, if a programmer can adjust software he needs without following the strict rules of a functional organization.

Designing an organization that specially fits software platforms helps to increase an organizations performance, Leite et al. (2020) found out. They compared 4 types of organizations by conducting about 20 Interviews with people working in those organizations. The team compared functional (siloed) organization, DevOps, Cross-Functional teams and platform teams. They not only found out that the platform team deliver a higher output, but also that the time-to-market was decreased.

From an organizational point of view, the concept and software developed in this thesis helps to measure the performance and workload of an organization. This consequently might help to deploy an efficient platform organization, transforming an functional organization and improving performance of an overall organization.

2.3 Technical

The third important fundamental to fully understand the reasons and impacts of this thesis is to understand, which different programming paradigm exist, what inner source development does and how it differentiates from traditional programming paradigm or open source development.

2.3.1 Open source paradigm

To understand how inner source works, it must be shown at first what open source development is, what the key factors are and which impact it has.

Open source is a paradigm, which the Open Source Initiative (2007) not only defined as open access to the source code of a software, but made concrete with ten criteria which must be complied to be called open source. Most important to understand inner source development are:

- Free Redistribution: The Software must be free to redistribute (Like selling or giving away) and free to combine with other programs
- Source Code: The source code must be accessible.
- Derived Works: It must be allowed to modify and derive own work of the given program

Feller and Fitzgerald (2000, pp. 59–60) described availability, modifiable and unlimited in usage as some of the criteria that must be met to be defined as open source. Stol et al. (2014) added some practices commonly observed in open source: Peer-Review, self-selection and frequent releases are also mentioned as typical for open source development.

In practice, these criteria are often implemented using public repositories, where a single user can choose a project they like, adapt it (Commercially or privately) and might channel back improvements they made for public use.

2.3.2 Inner source paradigm

Inner source as paradigm is based on the principles of open source. Stol and Fitzgerald (2015, pp. 60–61) described it as organizations adopting open source practices internally.

However, as open source was above especially emphasized as a way to open program code up to the public, not all practices commonly used in open source development are used for inner source development. Stol et al. (2014, p. 3) extracted, that the key difference from inner source to open source is, that the software is getting kept proprietary. Besides that, there are some elements of open source which are commonly used and define the key concept of inner source. Stol et al. (2014, p. 5) defined those as the peer-review (within an organization), communication channels, self-selection, and frequent releases.

Adopting these practices is bringing many benefits with it, as Stol and Fitzgerald (2015, pp. 60–61) showed:
Making written software internally (e.g. with a central repository) available makes reuse easier and therefore produces less development costs. Additionally,

it is more likely that bugs will be found, and that innovation will be accelerated as more people with different expertise have a look at the software and get ideas of how modules can be reused to solve other problems. Lastly Stol and Fitzgerald (2015, pp. 60–61) argued that inner source also improves mobility of personnel, as they get more used to different projects.
Capraro (2020, Appendix A) also found out various benefits that come with inner source. Besides the already mentioned ones, others like higher employee motivation and enhanced knowledge management were added.

If open source and inner source development are compared, it can be seen that both paradigm are close related to each other, even though open source works in a public context where source code is available for everyone's use and inner source just internal of a company. The core ideas of making source code available, modifiable, and adoptable to more people to increase performance, stability, and transparency are the same. The benefits of inner source development are not only the technical but also on the management side, as increased personnel mobility, and performance (mentioned above) are bringing an obvious business value.

2.3.3 Traditional programming in companies

The contrast to open- or inner source programming is the way companies are traditionally developing software.
Traditional software development has a close relationship to the organization forms explained earlier in Chapter 2.2. In this earlier chapter the difference between a functional and platform organization were explained. As stated there, a functional organization is inefficient for software development, as organizational boundaries need to be crossed. Therefore, splitting coding on one software (platform) into several functional entities reflects the way companies without a platform or inner source organization developing software.

Consequently, the inner source paradigm explained in the previous chapter is correlating with the platform organization explained in Chapter 2.2.2. The previous chapter looked at the organizational point of view and how the inner source paradigm can change companies organization. Now we saw open code sharing, modifiability and peer-review as a key factor from a programmers point of view. Therefore, inner source influences both: the business and technical workflow.

Being flexible and offering modularity is one important feature, the inner source paradigm offers (Stol & Fitzgerald, 2015, pp. 62–63). Assigning developers in contrast to a traditional development not only to one project but enabling them with inner source to contribute to several projects is bringing new problems with it.

Riehle et al. (2016) found out, that one big challenge with introducing inner source are the middle managers responsible for a certain business area or product. The

main reason is that developers putting work into software modules not belonging to their department do not directly contribute to their performance goals. Therefore, introducing inner source means for them losing resources, even though they might profit from it in the long run.

Measuring inner source development, analyzing cross-functional flow of workforce and assigning a value to it can be the basis for accepting inner source development not only by developers but also from middle and upper management.

2.4 Inner source and transfer pricing dependencies

In the previous chapters the three important foundations for Inner Source Transfer Pricing were set. The first main topic discussed were the basic principles of transfer pricing (Chapter 2.1.1), why it is important for companies (Chapter 2.1.2) and which methods exist to calculate them (Chapter 2.1.3). Additionally, this work looked more detailed at how full absorption costing is working, as this method is one important part of this thesis(Chapter 2.1.4). The second foundation were the ways an organization can be set up (functional, divisional and platform teams) to best fit the companies needs (Chapter 2.2). The third foundation were widespread paradigm(Open source, inner source and traditional ways) used to develop software (Chapter 2.3).

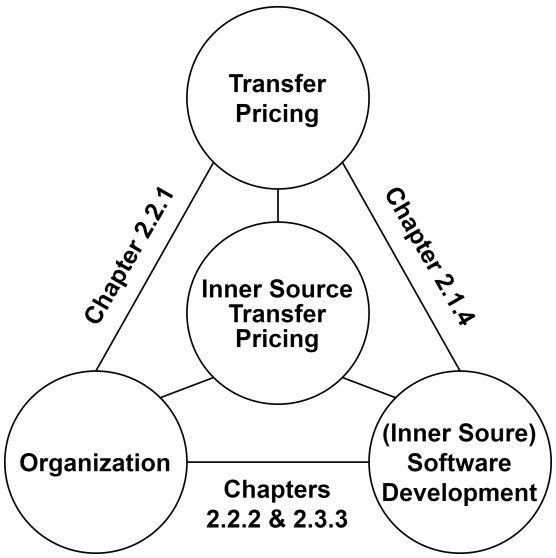
These three topics were discussed relatively independently of each other so far. The connections between the three main topics were considered pairwise up to now(See Figure 2.5). The relations already looked at so far, where those between the organizational principles and inner source development (Chapters 2.2.2 and 2.3.3), between the organizational principles and transfer pricing(Chapter 2.2.1) and between cost accounting/transfer pricing and software development in general (Chapter 2.1.4), but not specific to inner source. To fully understand the need and impact of Inner Source Transfer Pricing all three areas into are needed to be taken into consideration, which will follow in this chapter.

2.4.1 Problem awareness

In this section the previously detailed explained foundations are brought together to explain the need for Inner Source Transfer Pricing.

The previous chapters showed that inner source is a programming paradigm, where individual people are working on more than software modules or programs within their (superior) organization. It was discovered that programmers are crossing organizational boundaries, if non-platform organizations like functional or divisional are used. The Riehle et al. (2016) research discussed above showed,

Figure 2.5: Dependencies and text references between Inner Source Transfer Pricing topics



that these cross-functional developments can cause problems with middle managers. Additionally, it can be conducted, that not only functional/divisional organizations are causing problems (where a middle manager is responsible for his products development), but also platform-teams as software platforms easily can be reused in several products belonging to different entities or countries. Therefore, Inner Source Transfer Pricing has a connection to organizational boundaries and the problem that comes with it.

The connection between inner source and organizational principles from a management perspective is just one reason inner source development needs to be measured and evaluated numerically.
The second main reason to evaluate inner source development is the connection to taxation. In Chapter 2.1 it was discovered that transfer prices are needed for transactions which are done between connected companies crossing tax boundaries. Moreover, it could be seen, that implementing inner source development in functional or platform teams leads to crossing team boundaries (Chapter 2.2.2). Therefore, it can be concluded, that inner source development is directly connected to calculating transfer prices and a need for measuring development exists. Chapter 2.4.3 will go more into detail about needs why transfer prices needed to be calculated for inner source software development from a taxation point of view.

Furthermore, there is not only a need for calculating Inner Source Transfer Prices from a taxation and management point of view, but also for more precise cost calculation for software development independently of its usage. Chapter 2.1.3 is

an overview, which methods may be used to calculate a transfer price.
As said there, in this thesis the Cost-Plus method is chosen, which is based on full absorption costing. In Chapter 2.1.4 the full absorption costing and its application to software engineering in general is mentioned. The chapter showed that the differentiation between direct and indirect costs is harder. Reasons for that are, that on the one side, personnel cost are directly correlated to the development, but on the other side, as soon as developers are working in more than one projects, the costs needed to be split between those projects. Assigning the cost split incorrect can make a major difference, as software development is mainly personnel focused and less hardware driven.
Inner source development specifically makes use of cross-functional and platform development (See Chapter 2.3.3 and 2.3.2), therefore calculating the right cost-share of the development costs for each product developed is important to conduct a reasonable full absorption costing and consequently receive an acceptable transfer price.

To sum up the different point of views and impacts of Inner Source Transfer Pricing, it can be seen, that assigning values to the inner source development solves mainly two problems: On the one hand there is the taxation point of view including the full absorption costing problem (specific looked at in this work). On the other hand, there is the controlling and management view including strategic, organizational and personnel planning problems. Hanken et al. (2017, Part B and C) supported these two points of views, even though not specific for inner source development.

The following chapters will go into more details about these two points of views and which broader motivation and impact the work done can potentially have for these areas. Chapter 2.4.3 will go more into the motivation why there is a need to measure inner source development for taxation and its broader use on the tax system. Chapter 2.4.4 will go into the details about the controlling and management view and how and why it can make organizations more efficient.

2.4.2 Prerequisites for using Cost Plus

Before the broader impact of Inner Source Transfer Pricing on taxation and accounting is discussed, a look at the reasons why Cost-Plus was chosen is necessary. Future work must show, whether the concept developed here is transferable to other transfer pricing methods.

Cost Plus is chosen, as there is a direct connection between the working time performed for and project and the cots used to calculate for absorption costing. The structure of the calculation needed to be performed suites the nature of the data given.
The Transactional Net Margin Method is looking at the profit and compares

the height of the margin to a basis. The working time here might be suitable for a basis, but further research might show, whether this is true or also other measurements are suitable (such as code complexity, number of developers, number/category/ depth of interactions between organizations over ticket systems etc.).
The Comparable Uncontrolled Price Method and Profit Split Methods in contrast do not have a direct correlation between hours worked and the transfer price, as the price itself or profit is compared. A larger database of transfer prices with meta information might help to find a correlation suitable for the model developed in this thesis.
The Resale Price method compares the price of the good sold. Assigning a transfer price for regular contract programming work can be done. However, as there is no dedicated market for software developed with inner source paradigm, a method to compare these markets must be developed first before the price can be assigned.

This thesis is not to decide, whether the Cost-Plus Method is applicable for all software developed, especially those with the inner source paradigm. Therefore, it must be assumed, that all the preconditions for using Cost-Plus are met and this method is valid for transfer pricing.

According to the United Nations (2014, pp. 213–226) Cost Plus is especially suited for use cases that comply with the following characteristics:

- Typically, Cost-Plus is used for tangible goods or services, with mostly assembling activities and simple services
- Low risks performing the service is assumed
- The customer of the transaction is much more complex than the service provider. The complexity is by function (e.g marketing, selling, coordination, giving instructions) and risk (e.g. market risk)
- Cost Plus is not suitable for "a fully-fledged manufacturer which owns valuable product intangibles" (United Nations, 2014, p. 224)

2.4.3 Tax view on Inner Source Transfer Pricing

As first view on Inner Source Transfer Pricing the tax view was identified. As already described earlier in Chapter 2.4.1 with inner source development, there is a transfer price problem need to be solved. Within the taxation point of view, it can differentiate between two types of problems that are or might be supported with the concept implemented in this thesis: One the one side, the concept (as lengthy mentioned before) helps to calculate the transfer price for the taxation purpose itself. On the other hand, it might support solving a taxation problem of digital business models in general.

Previously it was conducted, that calculating transfer prices for taxation is a problem. Krause and Pellens (2018, p. 160) are supporting this assumption, especially with the knowledge, that the marginal costs approach zero.

Additionally, transfer prices are also getting increasing attentions by officials like the OECD. The problems occurring with developed software are not directly correlated to the transfer prices calculated with inner source development. However, OECD (2017a) lengthy analyzed the problems occurring with digital business models in general and set up a plan how to tackle with tax avoidance in these business models.
Olbert and Spengel (2017, p. 5) also analyzed that problem: "In the near future, revising the determination of transfer prices is one of the key challenges in designing an administrable system of profit taxation with a minimum of distortive effects for digital business models".
Calculating transfer prices for inner source development might be part of a solution to take the digital business model taxation problems. Whether this is the case or not, future research must show.

2.4.4 Accounting and management view on Inner Source Transfer Pricing

The second main point of view on transfer pricing is the controlling and management view. This view considers the dependencies between the organizational aspects, management and value assigned to programming between organizational boundaries.

That calculating transfer prices can help make a company work more efficient was already described in the previous Chapter 2.4.1. Riehle et al. (2016) concluded that separating product units is causing inefficiency and middle managers are more reluctant against inner source. Carroll et al. (2018, p. 4) made more concrete, that measuring performance makes the middle managers decision making easier.

Consequently, assigning values to inner source development cannot only help taxation, but also be of real use within companies. Cooper and Stol (2018) described the challenges that occurred within Bosch's inner source introduction. They recognized exactly the same problems described in this and previous chapters: Quantifying inner source development for middle managers and transfer pricing for tax authorizations.

3 Conceptual model development

In the previous chapter the three foundations for this work and how they are correlated to each other were described. Additionally, it was motivated, why the need for Inner Source Transfer Pricing exists and which impact it might have on taxation, controlling and management.

Now we will look at the conceptual model which was developed to quantify work time spend for certain inner source projects. Not only will be mentioned, how the final concept is looking like(Chapter 3.5), but also which information are needed (Chapter3.2), what options are available to measure inner source development in general (Chapter 3.2) and how the concept development progress proceeded (Chapters 3.3 and 3.4). The idea is not only to show the result, but also the process behind the development, so that certain design decisions can be understood more clearly.

3.1 Background and information requirements

This thesis wants to solve the problem, how values can be assigned to (Inner source) software development. As already said before, software development causes high personnel costs and low hardware costs, through which the exact work time per project is relevant enough to influence the transfer price.

If developers contribute to different programs, the costs cannot be assigned to a single product. Therefore, the personnel costs are indirect costs and not direct assignable at the full absorption cost calculation. Consequently, the development time must be split accordingly to comply with full absorption costing and Cost Plus consequently.

The main task of this thesis is to find out, how the work time a developer spends is distributed to certain projects. The data basis used are information, which usually are created during the development anyway and are known with the organization.

Most information which will be used in this thesis are taken from a version control

system (e.g. Git). Main information source is a list, which files the users were uploading to the central repository. One upload, called commit contains the following information:

- The software or module a commit belongs to
- The file path(Branch)
- The name of the file
- The changeset: An id of the change, if more files are changes at the same time
- The timestamp (Date and time) a file was uploaded, including time zone
- The author of the changed/uploaded files
- Number of lines added
- Number of lines modified
- Number of lines deleted

In addition to the commit data, information about the organizational background is given:

- The list of (inner source) projects
- The organisational units within the company
- The organisation an author belongs to
- The hierarchy behind the organisational units

These data are all, that are given to extract work time per project as exact as possible. Out of this context, some questions are arising, which will be answered in the following chapters:

1. Which information is the right calculation basis for the work time?
2. How can the right amount of time worked be estimated with the given information and in context of the calculation basis?
3. How to handle commits, which have no connection to others?

The answers to these questions will then be brought together in an overall concept, on which basis the final implementation is based on.

The real-world sample data are taken from a development organization with about 400 developers over about 1.5 years. Overall, the dataset contains about 228000 file changes, which conclude to 28600 grouped commits. For the general structure it can be said, that about 12300(10300, 7200) of the commit times were not further then 6h(4h,2h) apart from the previous one. Moreover, there are 403

commits with no time context to others.
As the data used in this thesis are taken as real-world data from, they are assumed to be representative. However, as the following chapters will show, some aspects of the concept still might differ from use case to use case, as (for example) some type of commits might be more usual in one organization than in others.

3.2 Calculation basis options

In this chapter, the possible options which can be chosen as a calculation basis will be discussed. As the commit data are taken from a version control system, the main ideas to calculate work time are based on these data.

For further evaluation, additional information might be taken to investigate those possibilities. Additionally, it might be helpful for evaluation to write software which measures the exact working time per unit and project.

3.2.1 Number of hours

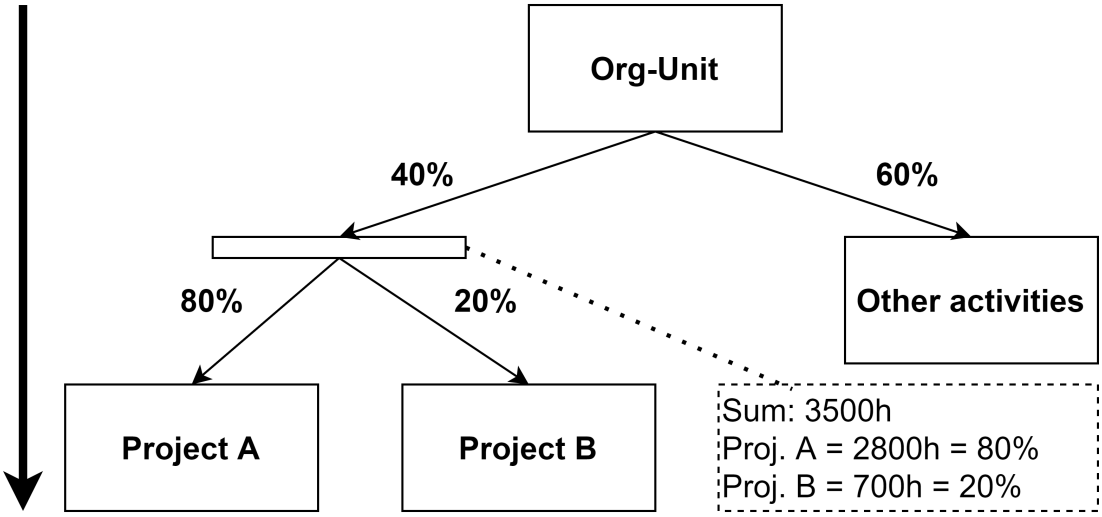
The first idea is to take the number of hours worked at each project and conduct the full absorption costing with these numbers.

The number of hours worked therefore must be calculated out of the timestamps of the commits. The basic idea is, to assign a work time to every commit made.

Advantages:
The main advantage of this calculation basis is, that it is an exact way of calculation, as every single commit, and therefore every time a developer is working, is included in the cost calculation. The assumption is, that a reasonable working time per commit can be estimated just with the data available, so that these data can be used for further calculation.
Having an exact working time in minutes (or hours) means, that each time spend on each project can be summed up. Out of this, a share can be calculated, how the costs must be split between the projects conducted.

Another big advantage is, that with the working time in minutes per project and the percentage-splits, we can do a top-down calculation and bottom-up validation of the cost calculation.

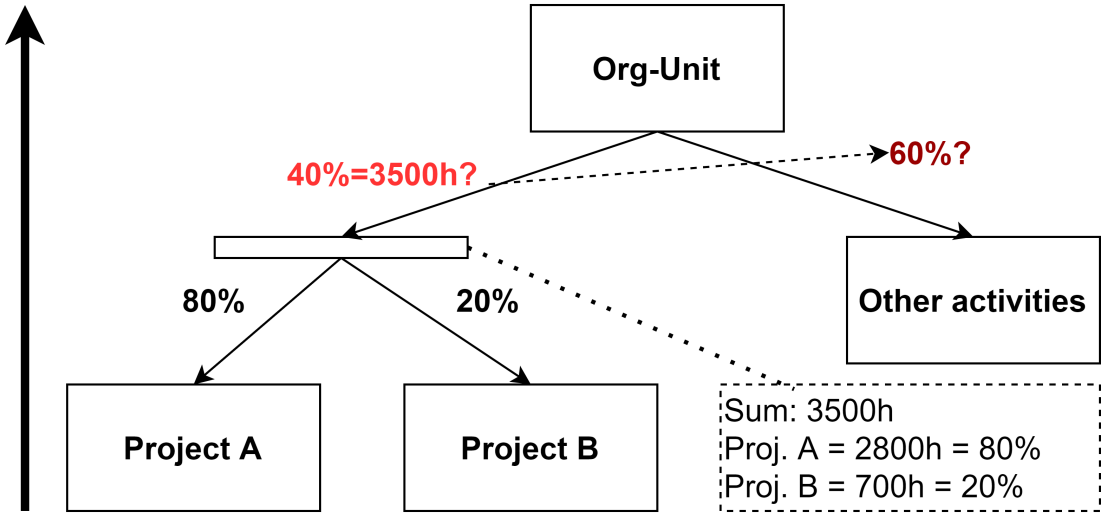
Figure 3.1: Top-Down software cost calculation example



The top-down calculation which can be done by assigning percentage-splits per project can be seen in Figure 3.1. The main assumption is, that the organizational structure is given and that it is already determined, how the costs are split in percent of the organization itself. In the example of the Figure, 40% of the overall costs within the organization are assigned to the projects included for the calculation. The other 60% are those projects outside the scope, which may be completely different activities (non-software development activities) or software projects, whose costs are determined on another way.
The further cost calculation is done by splitting the costs relevant to the project (40% in the Figure) again by the share determined by the working hour calculated out of the commit data. For example, if the commit data calculation (done in this thesis) results in 3500 hours overall working time for the two projects looked at, the costs might be split 2800 hours (80%) to Project A and 700 hours (20%) to Project B. These numbers then are used within the full absorption costing to calculate e. g. the personnel cost.

Furthermore, the absolute amount of working time can be used within a bottom-up validation (See example Figure 3.2).

Figure 3.2: Bottom-up software cost validation example



By knowing an estimated working time in hours (which includes every single commit) it is possible to validate, whether the assumed percentage split from the Top-Down calculation is valid or not. In Figure 3.2 it can be seen, what it means. Knowing, that the sum of the working time spend in projects A and B is 3500 hours, it can be verified, if the 40% of the working time spend in the overall organization is reasonable or not. To do this, the real working time (e.g. as an information from HR) is needed. Consequently, finding out a more reasonable way to assign the costs helps other departments with their cost calculation.

It can be seen, that working with a (as exact as possible) number of hours worked, is not only more exact with the cost calculation (conducted Top-Down), but also helpful for validating the way of calculation (Bottom-up)

Disadvantages:
As one major disadvantage of working with number of hours, the handling of single independent commits can be named. As inner source development is a flexible way of organizing the software development, not every commit must come from and elaborate member of the core development team. Developers who are often using, but just changing one or very view times the code of other software modules do commit irregular. Those commits have just a view or no connection to other commits. Therefore, calculating working time spend for exact this commit can be hard.

The biggest challenge when coping with numbers of hours worked are the irregularities in the commit structures. To find out a the time worked within a commit without having the real working time on hand, certain problems have to solved. The most important are for example the handling of the nighttime (When was

the developer really working), the midday break, the handling of greater time distances (e.g. days and weeks: Holiday, or just no commit?) and finally the question of how to deal with overhead like meetings.

Designing a dedicated software to filter the exact work time would make the calculation much easier. However, this thesis tries to calculate working time as reasonable as possible from the commit information at hand.

3.2.2 Percentage split

The second way, on which basis the cost can be calculated is to look at the percentage distribution of the commits only. This method tackles the problem described by using the number of hours as calculation basis: Including every commit, especially singular ones with no less time context can be hard.

The idea behind this calculation basis is, just to look at the percentages, how the work effort within an organization is split. This means, that not the number of hours is relevant and therefore less complex calculations have to be done.

Advantages:
The biggest advantage this method is bringing, is the reduction of complexity. Not only the calculation complexity can be made easier, but also the amount of commits which mus be considered, can be reduced.

Within a large organization, having a large number of commits might be enough to get an exact number, how the working time for the overall organization is split between several projects. Reason is, that with many commits included in the calculation, single ones do not have a huge impact on the percentage, as with the growing number of commits considered, the percentage-split is floating less. Further research and practical test must show, if just assigning percentage values might be exact enough.

Disadvantages:
The main disadvantage of just considering the percentage spit is, that the bottom-up validation might be inaccurate or not doable at all.
Depending on the structure of the commits and the commit behavior within an organization, a huge number of hours might be ignored. As looking at the percentage split only means, that no hour calculation is necessary, comparing the total hours spend in reality with the numbers calculated is not possible. Even if the working time is calculated for all commits included in this method, the comparison of the working time calculated to the true working time might be much more inaccurate, depending on how much single commits were ignored.

3.2.3 Lines of code

The third idea, how the costs can be assigned to the projects, is looking at the number of lines of code (LOC), a project was committed to. Using this method, it is assumed, that LOC do represent the amount of work that flowed into the module.

Advantages:
The biggest advantage using just the LOC is, that this information is already given by the commit. Therefore, applying this method is very easy to calculate with, as just a percentage split needs to be calculated.

This means, that (off course) the top-down calculation is possible. Additionally, a bottom-up validation is also possible, but not as extensive as with knowing all working time in hours (Chapter 3.2.1). It is possible to validate and find out, how much a certain developer (or business unit) is contributing to a software platform and validate this information with the planning and management.

Disadvantages:
One disadvantage using LOC as calculation basis is, that (as already suggested above) there are less possibilities to make a bottom-up validation. If organizations are doing more than just writing software, validating bottom-up is much more harder, as these performance cant be measured in LOC.

Additionally, it is not easy to quantify the relationship between LOC and work time spend, as a large conceptual phase can also result in a complex to understand, but small code. On the other hand, large amount of LOC must not mean, that the work effort was higher, as code can also by replicated easy.

3.2.4 Mixture

Another possibility which can be used as can calculation basis is mixing the previous mentioned suggestions. In this way, some of the important aspects can be combined, to find an optimum solution.

This means, that not only the number of hours is in focus for all commits, but also the possibility exists to handle different commits in different ways. Depending on the detailed concept, the results might be more or less exact.
Possible ideas calculating exact working time for some, assigning a projection for others or including LOC in the extrapolation.

The following chapters are concentrating on finding a way, how these aspects can be brought to together to make working time calculation more precise, without being to complex. It is important, that the concept developed is as generally applicable to other datasets as possible, without getting to inaccurate.

To bring several possible calculation basis together means in concrete terms, that for example an correlation between the LOC and time worked for a commit must be found. Additionally, a way of including the nighttime or holidays must be found.

3.3 Basic working hour calculation

As not a singular value (e.g. just LOC) is used as calculation basis, but several aspects are considered, a detailed concept must be set up, which takes together multiple influence factors. Main goal after all is still to calculate hours worked, so that these numbers can be reused for further cost calculation (full absorption costing) and top-down validation.

Before a detailed concept for calculating working hours can be set up, the structure of the commit data given must be analyzed and find out correlations which help to extract the time a developer worked on the software.

3.3.1 Time difference analysis

The first aspect that were analysed was not the time of commit, but the commit behavior itself. Idea was to find out, which commit regularity can be seen by looking at the time difference between two commits of the same developer.

3.3.1.1 First iteration: Time difference based LOC/H

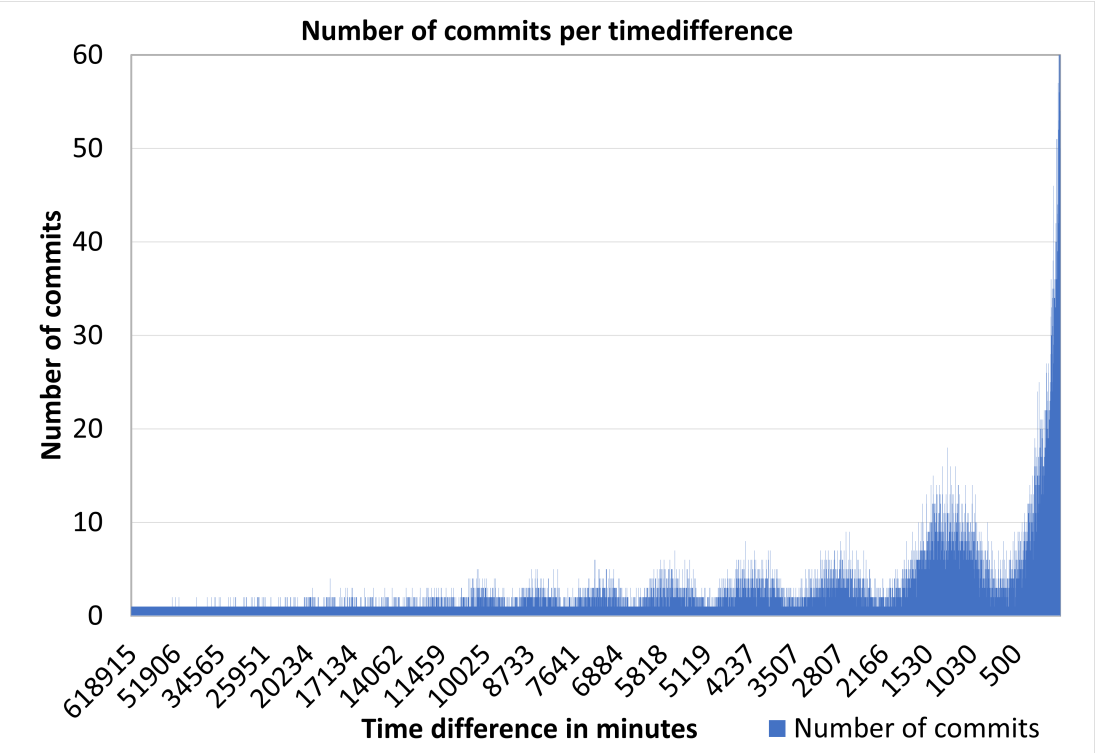
The analysis and first value calculation concept was sketched by answering questions on basis of the data available.

The first question answered was:

How many commits have a certain time difference?

Idea behind this question was to find out if commits accumulate with a certain time difference. To analyze this, all the commits were grouped for each user and than analyzed, how much time (in minutes) they were apart. Ordering the time difference in an ascending order and plotting them leads to the results shown in Figure 3.3.

Figure 3.3: Number of commits per time difference



The Figure shows the first two interesting things:
At first, the daytime structure is immediately visible. There are a low number of commits made in a 12 hour rhythm, which is why there is a valley in the graph at around 720minutes(=12 hours) and all consecutive 24 hours(e.g. 36 hours=2160minutes and 60 hours=3600 minutes). These observation seem obvious with the background of typical workday being less than 12 hours. The second interesting aspect which can be seen is, that most of the commits are done within 12 hours. The number of commits per day is decreasing, the more apart the days are. The last important and visible spike is around 7 Days (=10080 minutes). Therefore, it can be said, that most of the time, developers commit rather regular within one day, but also mainly once a day is possible. To commit every 2-7 days seems is done less frequently.

This behavior gives important data of how different commits might be treated. One idea therefore might be to calculate every minute as work time, for all commits which are a certain threshold, but with one day apart. The threshold e.g. might be 6 hours or the whole workday (12 hours apart, until the valley of the graph in the Figure is reached). However, with that first attempt, the problems of night times and midday break are still remaining.
Another unsolved problem by using all commits within a certain threshold is,

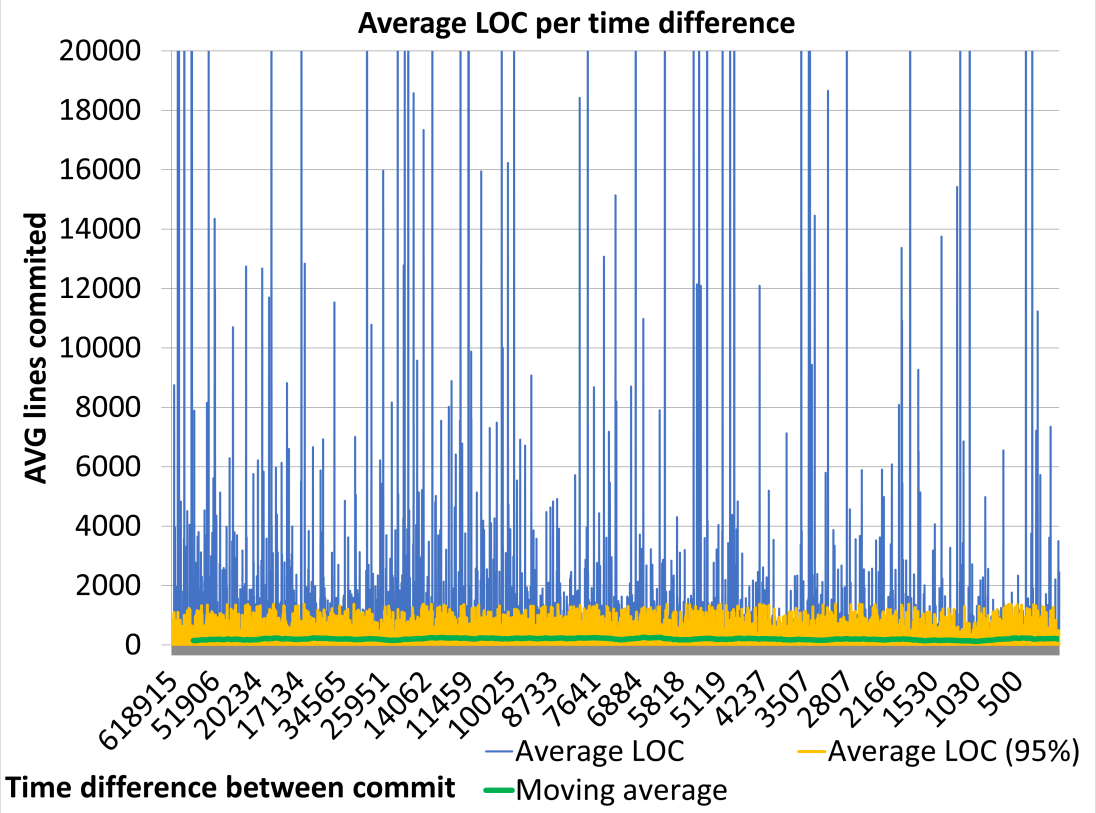
that all commits above this threshold are not used, especially those more days apart.

This construct leads to the next question to be answered:

Do commits, which are larger apart, contribute more LOC?

This question was answered by grouping all the commits by time differences and calculating the average number of LOC contributed at each time difference. The result can be seen in Figure 3.4.

Figure 3.4: Average number of LOC per time difference



In this Figure it can be seen the average lines committed for each time difference. The first interesting aspect is, that the number of LOC has a high fluctuation, as many commits are with less LOC and some commits contribute a large amount of LOC.
The reason for this lies in the structure of the data given. The software platform which is developed here consists of several modules, which sometimes are put together. Moreover, sometimes other large parts of code are either transferred or normally committed. This results into a commit structure, where (if the commits are ordered by LOC ascending) at some point the LOC are exponentially rising.

The largest commit in the database for example commits 2.6 millions LOC, which certainly is not representative for an average performance of a single developer. These large commits are the spikes which can be seen in Figure 3.4.

To not distort the picture with some smaller commits, the average LOC per time difference was calculated again, but with only the smallest 95% included. This results into the largest commit being about 1200LOC and therefore ignoring all the large single commits.
The result can then be seen in the Figure as a relatively stable value over all time differences. Therefore, it is not relevant, whether the commit is one or seven days apart, the commits always contribute about the same number of LOC. The moving average of the 95%-LOC is even more stable.

Knowing the commit structure from a time difference point of view, the next question raised, and **first concept** was created:

How can working time be calculated based on the time difference?

The idea behind this first concept is, to calculate with the time difference structure as background knowledge a LOC per Hour (LOC/H) performance value, which then can be applied to all the commits taken into consideration. Every commit, which is less than 9 hours after the previous commit from the same user, was calculated fully, as a temporal connection can be expected (Only view commits with 12 hours difference, see Figure 3.3). For all commits larger than 9 hours apart, it was assumed, that one workday is 9 hours long. Therefore, all commits with a larger distance must be broken down to assume only 9 hours of work and the rest of work free time. This was done with the following formula:

$$Avg_{\frac{LOC}{H}} = \frac{LOC}{mod\left(timediff, \frac{timediff*900}{1440}\right)} \tag{3.1}$$

The explanation of formula 3.1 is relatively easy: 9 hours working means, that 15 hours of a day (=900 minutes) was not worked. This formula looks at every time difference and breaks down the share for each 24h time slot. Example: A commit with a time difference of 40 Hours (which extends to almost two days). This can mean (in theory) two extremes: 18 hours working, 22 hours not working but also the other extreme with 10Hours working and 30 hours not working. This formula applied here simplifies it and assumes, that 9 hours were worked full in the first day. Unknowingly 16 hours were in the time slot of the next day. 16 out of 24 hours are 2/3. Therefore, 2/3 of the 9 hour workday is simply calculated, which are 6 hours of work in the second day. Finally, this leads for the 40 Hour workday to 15 working hours.
However, this is obviously not perfect, but gave a good first look how the working hours are correlating with the time differences.

Figure 3.5: Average number of LOC/h per time difference

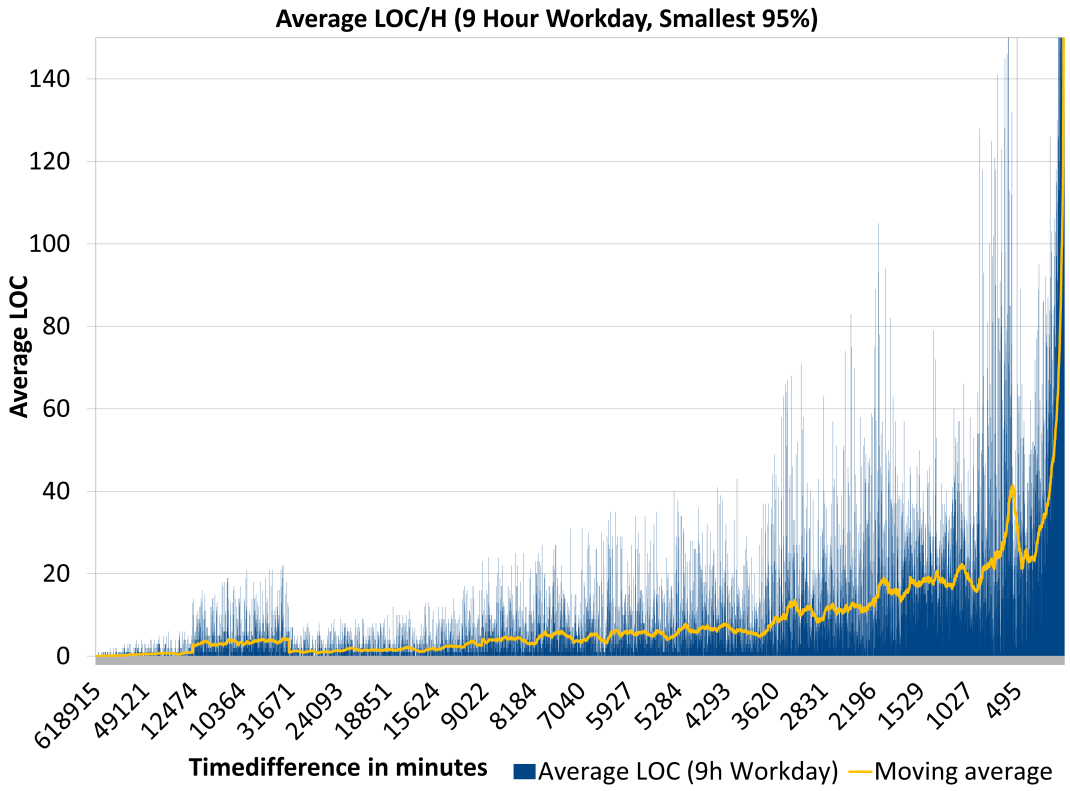


Figure 3.5 plotted the calculation from formula 3.1 for every time difference for all commits within the 95% range. The first obvious observation is, that the LOC/H performance is increasing as the time difference between the commits are getting smaller. Besides that, the LOC/H is relatively stable within each day-frame (e.g. 720 - 2160 minutes), except the commits <1 day. The moving average is decreasing slowly for every day larger apart. There are two main reasons for this: On the one hand, the formula is handing 24h-blocks. On the other hand (and more importantly), looking at commits which are for example two or three days apart assumes here, that two or three days were fully worked.

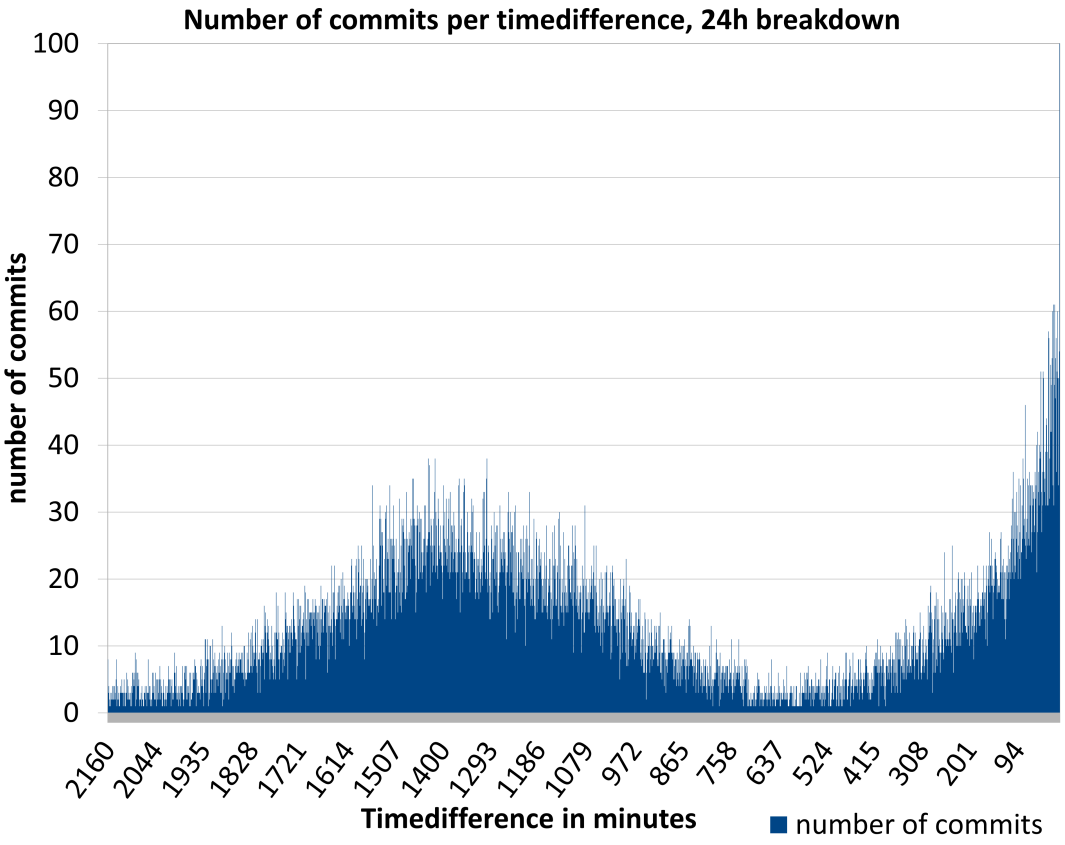
3.3.1.2 Second iteration: 24h time difference based LOC/H

Taking the analysis of Figure 3.4, it can be seen, that commits are following the day and week structure (Daily commits, 2-day-commits, 3-day-commits etc.) However, in Figure 3.5 it can also be seen, that every commit, whatever time distance to the previous commit existed, roughly committed about the same LOC. Consequently, it can be said, that even commits larger than one day apart are not having e.g. double the work time spend. Therefore, it is common for developers to commit at least once per day and most of the commits further

apart might be days like weekend or holidays. This behavior can be confirmed by looking at holidays and weekends of the sample data, even though not all commits stick always true to these dates, as the developers are working internationally.

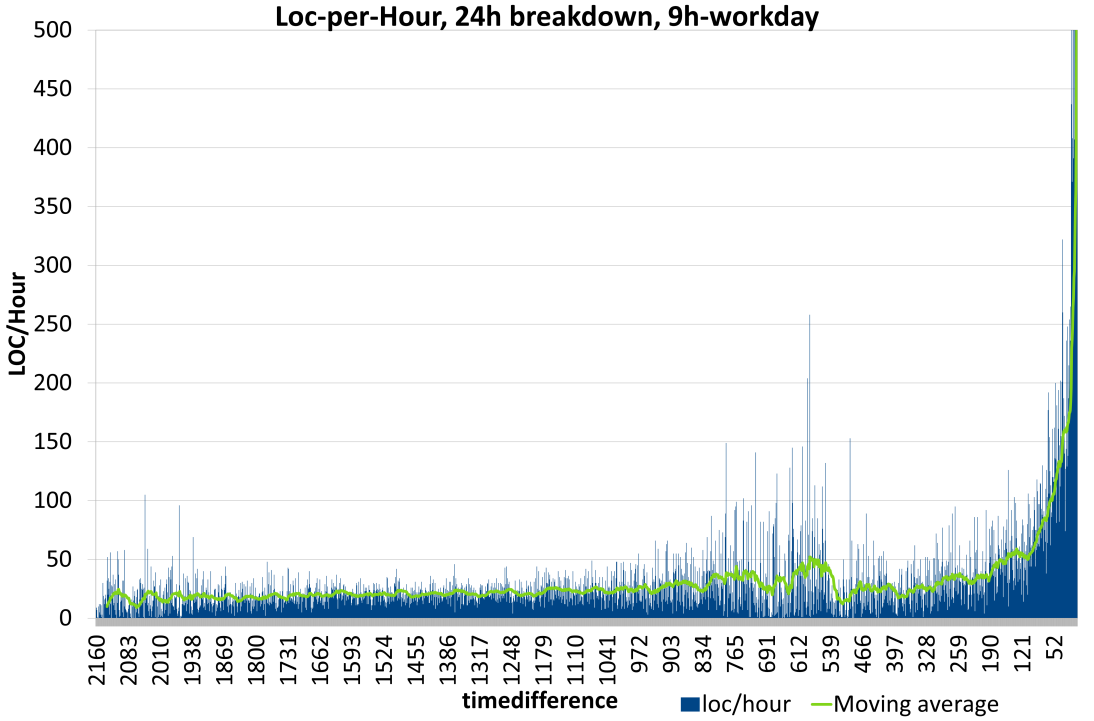
On this basis, the first concept was redesigned at a **second iteration**: All commits were broken down to a time frame, which fits within the daily commit structure (36 Hours max). The result (Figure 3.6) shows, that still a larger amount of commits are within one day, but many are daily with a steady curve spiking at 1440 minutes time difference (24 hours).

Figure 3.6: Number of commits per time difference and 24h



Appling the formula used above again to all commits with 24h-breakdown leads to the plot in Figure 3.7. In this graph it can be seen, that with assuming daily commits, the LOC/H performance is relatively steady for all commits larger than 12 hours apart. All commits less than 12 hours apart are having a much higher LOC/H performance. Most relatable reason for the exponential growth towards small time differences are copied software parts, which have much LOC, but just take a view minutes to commit. Consequently, those copy and paste commits let the LOC/H performance exponentially grow.

Figure 3.7: Average number of LOC/h per time difference and 24h



The moving average of the LOC/H performance is first good guess, which later might be used to assign working time for all commits.

Problems:
Even though the first concepts showed a stable LOC/H performance which might be used, there are also some problems coming with it.

Most importantly, the formula applied guessed the working time based on a percentage for each time difference. Therefore, all commits which are equally far apart assume the same work time, not looking at the time and circumstances a commit was made in.

To solve this problem, not only the time difference was taken into consideration, but in the following chapter, the time of the commit was looked at.

3.3.2 Commit hour analysis

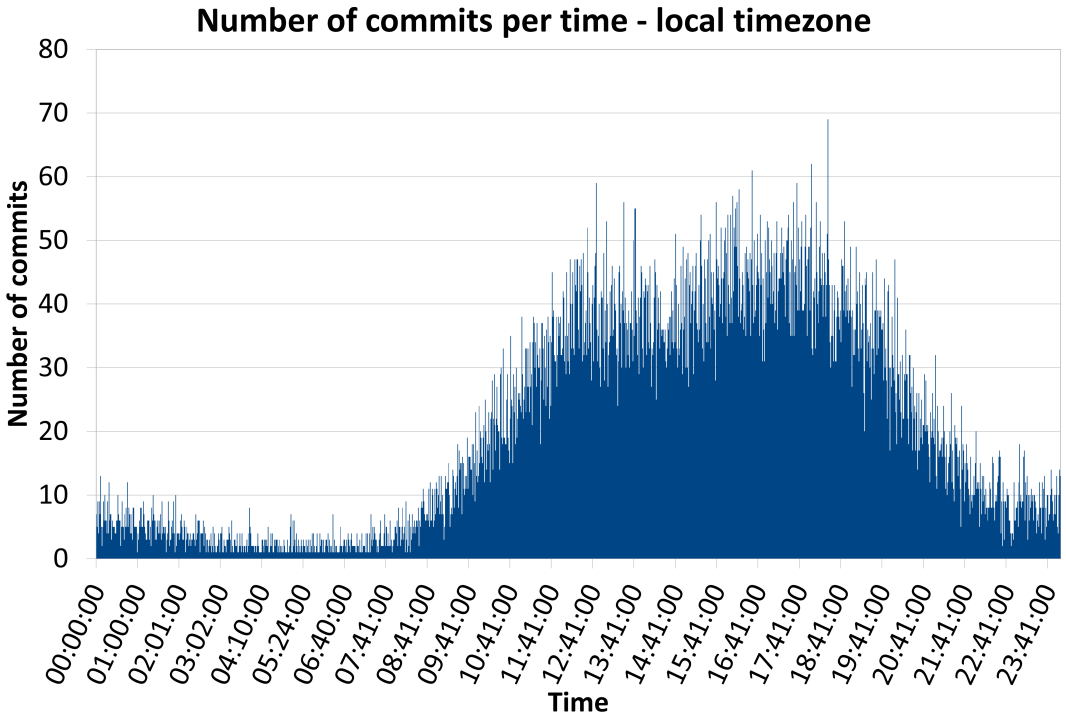
3.3.2.1 Third iteration: Timestamp based LOC/H

For the **third iteration** the time - to be more precise the hour and minute- of the commit was analyzed. The idea here is, that the LOC/H performance is guessed with the background of knowing the daytime the developer worked. Analyzing

this structure leads to a commit distribution by hour of the day, which can be used to guess a reasonable work time.
The first question to analyze the working time by commit hour was:

At which times to developers usually commit?

Figure 3.8: Number of commits per time



Many aspects which can be observed at Figure 3.8 are as can be expected: During the night, only a low number of commits are done (single-digit numbers), even though there are some. The number of commits is increasing in the morning and decreasing at the evening, with a small dip during midday. It must be noted, that during the afternoon comparable more commits are done than in the morning. Additionally, there is no straight 'cut' in the evening (like classic 9-to-5 office hours), but the workload is decreasing steadily into the evening.

This distribution leads to the first important finding for this chapter: Personnels work time is different from developer to developer, there cannot be simply assumed a fix working time, on which basis the working time can be calculated on. Therefore, the next step is to find out if there is any connection to the previously looked at time differences. The question now to be asked is:

How do the commit time and time difference to previous commit correlate?

Figure 3.9: Number of commits and time differences per time

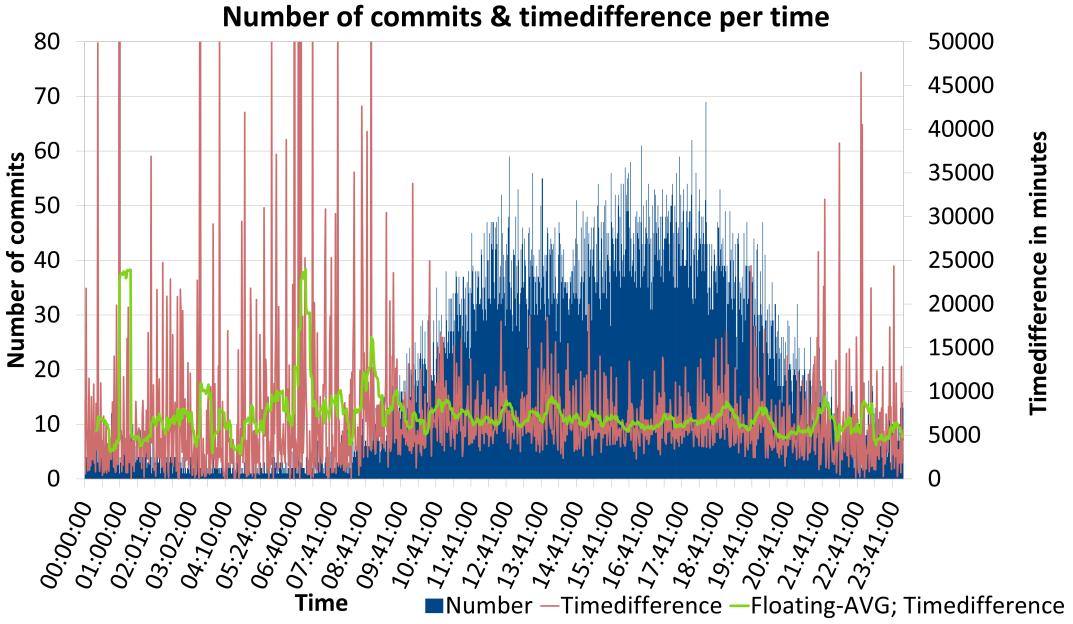
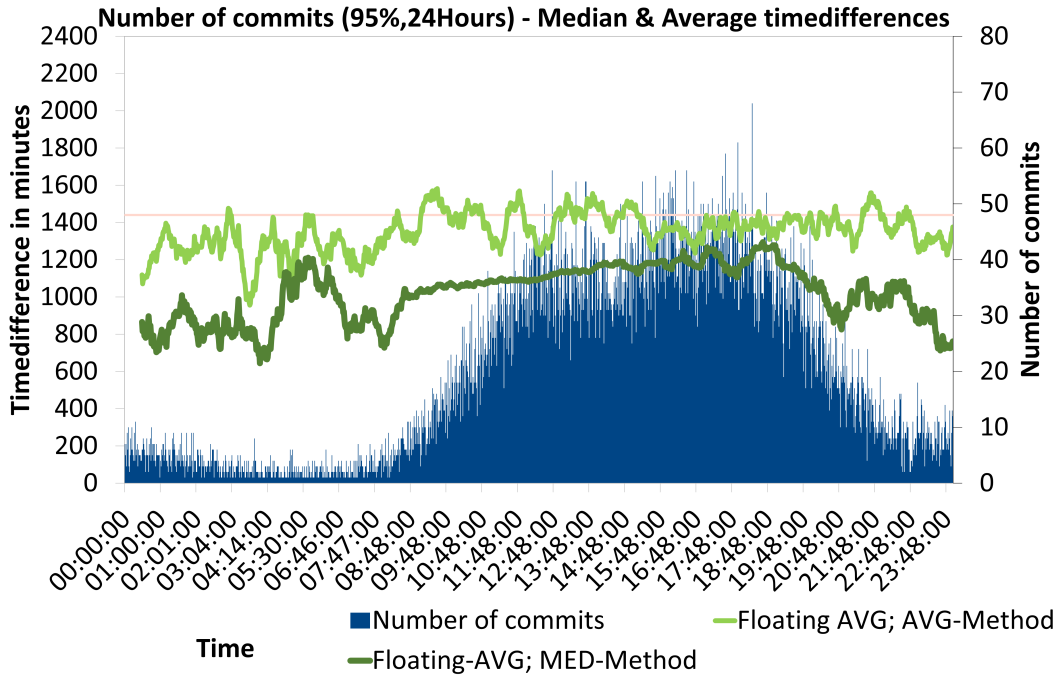


Figure 3.9 shows on the left y-axis the number of commits per time(See Figure 3.8). New is the secondary right y-axis, which shows the average and floating average of the time differences of the commits at the respective timestamp. The time differences here are the total time differences without breaking them down to 24h.

Analyzing the results, the time differences are smaller within the day and larger over night. The fluctuation of time differences outside usual working hours is high, which might be connected so single irregular commits, which are automatically planned to be executed over night. However, this should be evaluated again in future research, as this data set gives no hint which commits truly belong to automatic commits.

As with the time difference analysis before, the graphs were done again, but this time with breaking down the time differences to a 24 hours basis and just including the smallest 95% of the commits. The results are shown in Figure 3.10, with being the time differences on the left y-axis and the already looked at number of commits on the right y-axis.

Figure 3.10: Number of commits and average/median time differences per time



The two lines are showing the moving average for each time difference value calculated at each timestamp. These lines are differentiating in the way, the time difference value for a single timestamp was calculated. The lower line is taking all the commits of a timestamp and calculates the median time difference. The line shows the moving average over all median time differences. The line above takes all commits of a timestamp and calculates the average time difference.
Comparing these two lines it can be seen, that while the average time difference is floating during the day around 1440 minutes (24h), the median is steadier during the day, but lower. The median time difference line declines in the evening hours (about 19:00 o'clock /7PM) and increasing in the morning(8:00), which might hint to a working behavior of working/committing steady during the day and a first/final morning and evening commit.

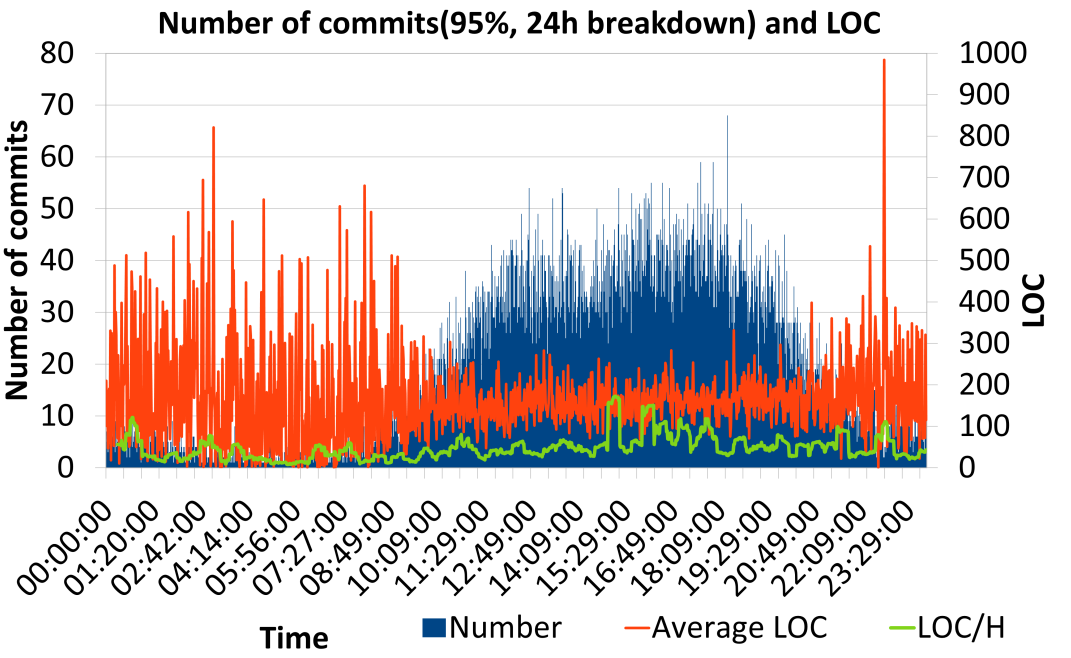
The commits over night are a lower number, with larger floating dime differences as already mentioned. This leads to a high fluctuation during the nighttime. Looking at this numbers it can again prove an already made guess right: Looking at time differences as only criteria for development performance is not right, as for almost every timestamp(especially during the day), the structure behind the commit differences is similar. Taking only the time difference as criteria would lead to similar LOC/H performances, even though one commit was made in the morning and another one after a whole day of work.

Out of this insight, the next questions raised:

How do the commit time and number of LOC correlate?

To answer this question, for every timestamp, the average number of LOC committed were calculated. Figure 3.11 shows the plot of this analysis.

Figure 3.11: Number of commits and average LOC per time



In the chart in Figure 3.11 it can be seen, that during the day a low number of LOC is committed. During the night, larger number of LOC are contributed. Over the night, the fluctuation is much more higher than during the day. This figure also shows another way how the LOC/H performance might be calculated: Taking the average LOC committed per time difference as basis and calculate with the time difference analysed above a LOC/H performance. As to expect, the LOC/H performance is on an almost similar level. Interesting to see is, that the LOC/H Performance is slightly lower over night, as there the time difference between the commits is larger.

However, as with previously introduced possibilities of calculating a LOC/H performance, this iteration might not be a valid enough, as the daytime plays a secondary role.

All the methods how working time can be measured were based on the principle to calculate a LOC/H performance and use this to calculate working time. Therefore, next step is trying to calculate a more reasonable LOC/H performance out of the working time determined.

To make this possible, first of all the working time - which is ultimately tried to be calculated - must be calculated out of the LOC/H determined before. As this is a circular reference, the only way out is to find a method of finding working time without looking at LOC/H. It must be noted here, that it might seem pointless in the first place to calculate working time (which is the main goal of the overall concept) and then to calculate LOC/H out of this after all, if we already got the working time needed. However, as stated in Chapter 3.2.1, in inner source development not every commit of a project must come from an elaborate member of a team. There might be a large amount of commits with less or no connection to other previous commits. Consequently, those commits cannot be analyzed with any working time method using the time difference. Moreover, large commits were already filtered out as they are distorting the picture. For those two types of independent commits, it is viable to use the LOC/H performance calculated through other commits and assign the working time over this method.

Interim summary:

To have a better overview what is already known and which knowledge can used to build a working time measurement model, a brief interim summary is needed.

It could be seen, that during the day, the more regular (stable time differences), but smaller commits (Less LOC) are made. During the night, a view larger commits with larger time differences are made. Therefore, it can be concluded, that normal working times are (as to be expect) over the day, with irregular work being committed over night (e.g. through automatization).

Moreover, different approaches how a LOC/H price can be calculated were shown: Using the LOC and time differences with the timestamp point of view at first and using only time differences secondly (With and without 24h-breakdown).

Looking at the results it can already be seen, that overall speaking, each commit seems to be more or less equally productive.

3.3.2.2 Fourth iteration: Working time concept

To fulfil the need of measuring working time based on time difference to the previous commit and the timestamp it is committed, a working time concept was created. As before, a basic working time of 9 hours per day was assumed. The concept was tested and implemented with this work time, nevertheless it can be adjusted easily in the prototype if needed.

The concept developed differentiates between four base cases. Each base case is differentiated by the time difference of the current commit to the previous one: First of all, those $<$360 minutes (6 Hours), secondly those $>$720 minutes(12 Hours) and $\leq$2160 minutes(36 Hours), at third those $\geq$360 minutes and $\leq$720 minutes and lastly all $>$2160 minutes.

Main idea behind the differentiation is to take the contexts (e.g. nighttime) behind the time differences into consideration. On the one side, there are those commits, which happen after holidays, weekends and after long time without commit. Commits with >2160 minutes (36 Hours) time difference are treated as irregular commits, as the time distance is larger than usual workday apart (and the previous analysis showed, that daily commits are usual). On the other side, there are those commits which happen within one day (36 Hours limit). All commits less than 36 Hours apart are again differentiated by time difference, as a commit less than 360 minutes after the previous one is less likely to have only a small working time. For example, a commit 15 hours after the previous one is more likely to have a nighttime in this time period than a commit which covers the day.

Commits <360 minutes time difference:

In this concept, all commits which are <360 minutes in time difference to the previous commit will be counted as full working time. Background and main motivation is, that 6 hours seem a reasonable time difference where person might have worked on project and not leaving work. In the sample data, about 7440 commits (26%) are in this group.

Of course, this approach cannot guarantee, that the whole time was worked on the project the commit contributes to. Therefore, this concept assumes that the whole time is spend on the project, as no other information (e.g. concrete file change date, ticket systems, mailbox timetable) are available. This approach relies on the effect, that for large data sets, the average time spend per project is correct over all projects in the long term, but single commits can be wrong assigned. Especially for developers whose development work is almost fully included in the measured dataset, assuming full work time also accounts for organizational overhead of the projects. In full absorption costing all costs need to be included, therefore it is not allowed to exclude overhead like meetings from the calculation.

Calculating every small commit as full working time is supported by the findings of Chapter 3.3.1 (especially Figure 3.6), where it can be seen, that commit regularly within smaller time differences makes a large part of the commits and is one common commit behavior.

Commits >720 minutes and ≤2160 time difference:

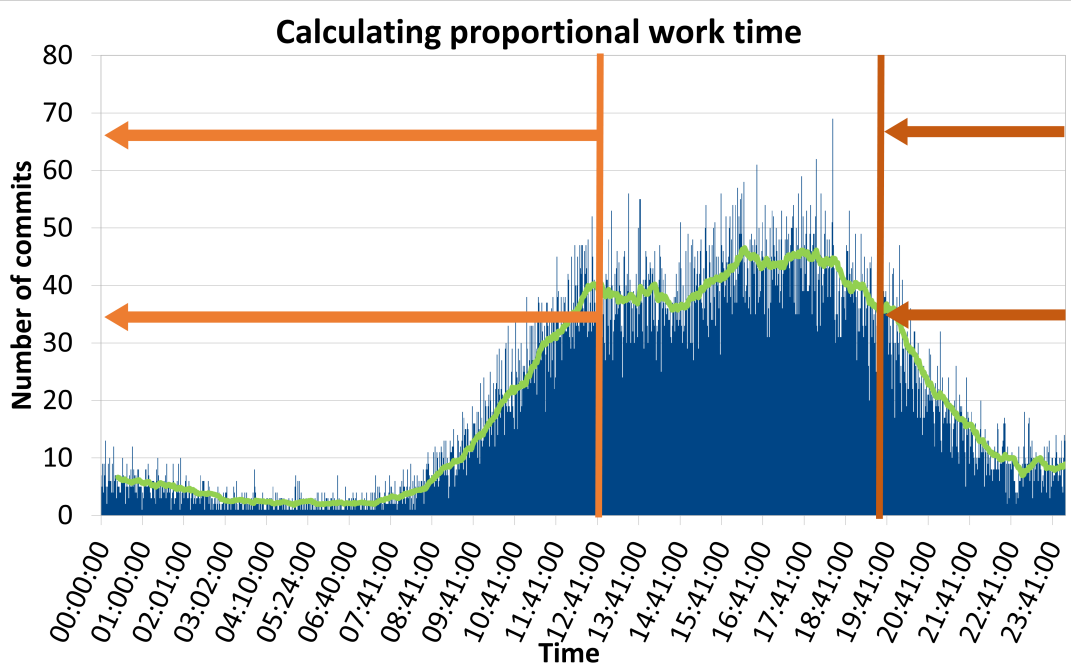
All commits which have a time difference >720 and ≤2160 minutes to the previous commit (daily commits) are calculated not fully, but in proportion to the typical sum of commits done in the time span they cover. In the sample data, about 6500 (22.7%) of the commits have this time difference. 12 hours was taken as a threshold, as for this time span it is very much likely, that the commit is through the night with no work time there. Having work time larger than 12 hours during the day for example means, that the developer starts at 8am and finishes later

or equal to 8pm.
Due to the structure of the usual commit behavior (most commits during the day) and the proportional calculation method (which will be explained more detailed in the following), not the full 12+ hours are granted, but only those number of hours, which a typical developer is working(committing) through that time.

Proportional work time calculation:

The proportional work time calculation is based on the number of commits which are done at the timestamp of the commit which is currently looking at.
The basic idea is, to start from the timestamp of the current commit and to go backwards in time until the previous commit is reached (See Figure 3.12). For every minute, the number of commits historically made during this minute of the day is summed up. Consequently, after iterating through 24 hours, the sum of all historically made commits is reached (In our sample dataset it is 28601).

Figure 3.12: Calculating proportional work time



After summing up the number of commits historically made through the time period of the commit, this number is set into proportion with the overall number of commits made in 24 hours. This proportion is then used to calculate the working time. If, for example, 75% of the historically typical number of commits where covered, it is assumed that the developer also performance 75% of the typical workday. This percentage is then used to multiply with the duration of a typical workday. In hour example 9 hours(540 minutes) were assumed, which

results by 75% to 6 hours and 45min workday (405minutes).

The consequence by calculating with the proportional method is, that a commit, which is exactly 24 hours after the previous one, get the full workday assigned (e.g. 9 hours). Two commits which are larger apart get a bit more, those less apart get proportional less working time assigned.
For commit near the 12 hour threshold this method seems not perfect in the first place, as commits 12 hours apart during the day (e.g. 8am to 9pm) have less than a full workday. At this point, the still unsolved problem shows that it cant be known if the person was really working 13 hours or having a large break in between. For this thesis it cannot be said more in detail which time was development time, break time, free time, organizational overhead or other projects. Therefore, future research has to fill in the gaps to filter enable filtering these time gaps and to make this model more precise.
Despite that fact, the proportional work time represents a historically typical work distribution. Consequently, applying the proportion to a large dataset might lead to inaccuracy for single commits, but confirms the overall trend, which is more important than correct values for every single commit.

The formula how the proportional work time can be calculated can be seen in compact in the following formula:

$$worktime_{proportional} = \frac{\sum_{i=timestamp_{currentCommit}}^{timestamp_{previousCommit}} \text{numberOfCommits}_i}{\text{numberOfCommits}_{overall}} * \text{worktime}_{day} \tag{3.2}$$

This formula describes the previously mentioned proportion: The sum of all historical commits ($numberOfCommits_i$) at the timestamps between two commits ($commit_{current}$ and $commit_{previous}$) in proportion to the overall sum of commits ($numberOfCommits_{overall}$). $worktime_{day}$ describes the number an usual workday has to have as a calculation basis, for example 9 hours. The result shows the estimation in minutes.

Commits ≥360 and ≤720 minutes time difference:

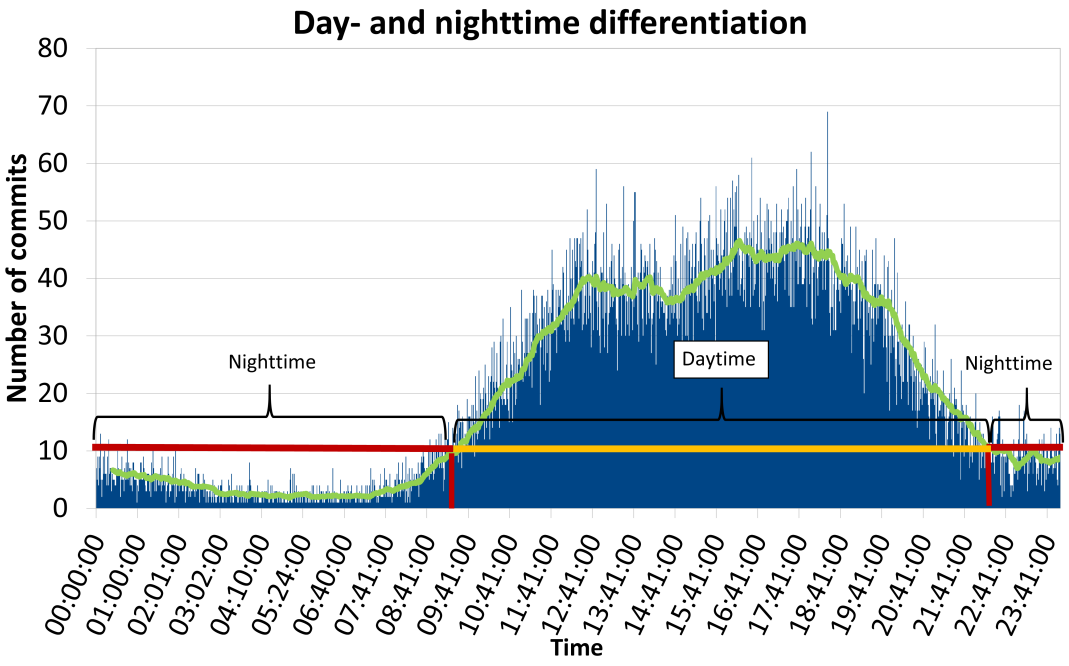
Commits which are between 6 and 12 hours after the previous one are treated differently depending on the timestamp, the commit was done. Background is, that it not sure for those commits, if they are just a longer workday without a commit or a short night. As the sample data show, both cases seem to happen. Especially commits in the afternoon more than six hours after the morning commit are not unusual.
The commits which are made e.g. once on the end of the workday have to be differentiated from those who happen during e.g. in the morning and were the

previous commit was sometimes over night. In the sample data 1350 commits (4.7%) belong to this group.

The easiest way to differentiate the two mentioned commit cases is to look at the timestamp and time difference a commit has. The basic idea is, that a nighttime is declared which is used to grant a commit less work time for this timespan. Commits which are covering less or no nighttime, but rather larger parts of the day are assumed to have more work time as those who cover the night.

The declaration, which time is nighttime can be made dependent on the historical commit data or just be fix values. The exact time where the night ends and begins can determined by a percentage limit (e.g. 15%) of the maximum number of commits a single timestamp has. To be not influenced by single timestamps where large amounts of commit happen, the floating average of the commit numbers is taken in the example of this thesis. The time in the morning and evening, where the limit is crossed, is set as end and begin of the day. Figure 3.13 shows visually how the limit can be understood.

Figure 3.13: Day- and nighttime differentiation



The main assumption is, that on commits during the day was worked less than on commits which go through the night. Additionally, morning-commits (previous commit was before or during the night) are getting not fully every minute as work time, but just proportional to the historical typical work time for this time period. If for example the night is from 10pm to 7am, the previous longest time a

previous commit can apart is 7:01pm. As it is very unlikely that the whole night was worked, only proportional work time is assigned.

The first step is to calculate, how much of the nighttime was covered through the commit:

$$nightshare = \frac{\sum_{i=timestamp_{currentCommit}}^{timestamp_{previousCommit}} x_i * 1}{\sum_{i=timestamp_{nightBegin}}^{timestamp_{nightEnd}} 1} \tag{3.3}$$

$$x_i = \begin{cases} 1, & \text{if } i \in nighttime, \\ 0, & \text{otherwise} \end{cases}$$

This formula counts, how many of the total possible minutes a night has (denominator) were also covered by the current commit looking at (numerator). The result (a percentage) is then used to calculate the working time:

$$worktime_{nightshare} = nightshare * worktime_{proportional} + (1 - nightshare) * daycoverage$$

$$daycoverage = \sum_{i=timestamp_{currentCommit}}^{timestamp_{previousCommit}} x_i$$

$$x_i = \begin{cases} 1, & \text{if } i \in daytime, \\ 0, & \text{otherwise} \end{cases} \tag{3.4}$$

The logic behind the working time calculation with nightshare (360 to 720 minutes time difference) is, that a commits gets as much proportional working time (see Formula 3.2) as it covers the night($nightshare$). This ensures, that commits which cover large parts or the whole night only get the work time for the night assigned, that historically was typical for that time. For the minutes which belongs to the day($daycoverage$, sum of minutes during the day), the height which the night is not covered is fully assigned.
Example: The night is between 10pm and 7am. Commit 1 is at 8am, the previous one at 9pm. As Commit 1 covers the full night ($nightshare = 1$), it is fully calculated as proportional. Commit 2 is at 4pm, the previous one at 6am. Commit 2 covers only 11% of the night ($nightshare = 0.11$), this amount is calculated proportional. The minutes during the day (7am to 4pm) are calculated to 89%. Comparing these two main use cases it can be seen that a commit which was done early in the day got less working time (Just the historical typical amount). Those commits which rarely go into the night got almost regular work time assigned.

3.4 Independent commit handling options

In the previous chapter, a concept how working time can be calculated was developed. Through the four iterations various possibilities could be seen to archive the overall goal. While most of the (former) iterations are based on calculating a LOC/H performance value which is used to calculate working time, the latest iteration brought an alternative direct formula how the time spend per commit can be calculated.

However, as already described above, not all commits couldn't be included in the previous analysis or cannot be analyzed with the resulted concept. Main reasons is, that some commits are more than one workday apart (>2160 minutes). Additionally, there are single commits from users which contribute to the software platform only once. The latter cant be calculated by using a time difference. Moreover, it must be considered from case to case, if large commits (5% in our example, mainly large imports) have to be evaluated extra, as they distort the trends the analysis are showing. In the sample data, 12867 commits (45%) are more than 36 hours apart, 403(1.4%) have are single commits where no time difference can be calculated. It has to be noted, that in our dataset there is a large portion of commits which is further than one workday apart. This is due to the structure of the project, as the software developed is (as typical for inner source development) a platform where multiple organizations are contributing, but rather irregular than daily. However, it is important to understand, that the concept developed in this thesis unfolds it whole potential (and is most precise) by applied to a large number of commits, even if some commits are not directly linked to the projects/products being measured.

To be able to calculate top-down and validate bottom-up (See Chapter 3.2), all (or most of) the commits need to be calculated. In the following, four options (and there development) will be presented, which can be used to complete the working time calculation. In the following, all commits with no time difference and those > will be called 'individual commits' or 'independent commits'.

3.4.1 Ignore individual commits

The first (and rather obvious) possibility is to ignore these individual commits. This means, that all single commits are not included. For the to large commits, an individual threshold must be introduced, which is based on the structure of the data. As the sample dataset also got commits included, where other projects were completely imported at once, the threshold was set to 95%, which results in the largest commit being about 1200 LOC. This value of course may differ if no imports and just regular commits are done.

Choosing not to include individual commits results into not being able to make an

exact bottom-up validation, if the ignored commits are representing a significant share of the overall commits. On the other side, the percentage split might still be valid for a large enough dataset. Advantage using this method is the easy implementation and logic that is applied to it.

3.4.2 Apply flat rate

Another possibility to assign a value to individual commits is to calculate them with a flat rate, a fix number of minutes/hours for all commits.

This method is also easy to apply and implement. However, it might lead to deviations in the percentage split (as the flat rate represents not the same percentage like the other commits) or false total hours worked.

It might be useful, if the project we are looking at is clearly defined and it is well known, where the individual commits come from. An example use case where a flat rate might be applicable is a project where regularly (e.g. each semester) internships are coding extensions for a platform, which get included once at the end of the project, if they meet the quality criteria.
For those use cases, where the flat rate is applicable, the key question is, which rate is to be chosen. However, this can't be answered generally as the answer is depending on the project.

3.4.3 Apply LOC/H calculation

A third option to assign values to individual commits is to choose one of the previously calculated LOC/H performance values and apply this to the individual commits. An overview of the possible LOC/H values that can be chosen for the sample data set can be seen in Table 3.1.

The table shows various combinations, how LOC/H values can be calculated. First of all there are two dimensions which differentiate each 2 possibilities. The first dimension is the decision whether to use the original data or to break them down to 24h-ranges (assume daily commits). The second dimension differentiates between all commits and the smallest 95% (to get rid of the large outliers).

During the iterations in Chapter 3.2 several possibilities were discussed. One possibility was the way of calculating the LOC/H value grouped by the time differences of the commits (Figure 3.7). Out of this array of values, either the median ($Diff - MED$) or average ($Diff - AVG$) can be used to calculate a single value. The same is true for the LOC/H value resulted due to the timestamp grouping (See Figure 3.11, resulting in $Time-AVG$ and $Time-MED$). The last option *List* is not grouped by timestamp or time difference, but just calculating the LOC/H value out of the unordered list of commits without context.

Table 3.1: LOC/H possibilities

	Percentage	Method	Median	Average	Max LOC/H	Min LOC/H
original	100 %	List	2.25	157.75	1188660.00	0.00
		Time-AVG	24.72	242.95	170520.00	0.00
		Time-MED	2.05	126.94	170520.00	0.00
		Diff-AVG	1.25	20.23	23298.74	0.00
		Diff-MED	0.57	4.64	2742.84	0.00
	95%	List	1.90	54.99	75120.00	0.00
		Time-AVG	17.88	44.66	2276.93	0.00
		Time-MED	1.70	4.52	549.47	0.00
		Diff-AVG	0.97	6.65	6348.68	0.00
		Diff-MED	0.48	2.68	960.00	0.00
24-H Breakdown	100 %	List	5.20	172.08	1188660.00	0.00
		Time-AVG	32.17	255.05	170520.00	0.04
		Time-MED	4.63	130.75	170520.00	0.04
		Diff-AVG	15.04	81.12	23298.74	0.03
		Diff-MED	3.45	9.59	960.00	0.03
	0.95 %	List	4.45	58.09	75120.00	0.00
		Time-AVG	21.41	47.80	2282.40	0.04
		Time-MED	3.97	7.29	549.47	0.04
		Diff-AVG	8.96	26.53	6348.67	0.03
		Diff-MED	2.95	8.24	960.00	0.03

The main takeaway from Table 3.1 is, that there are many options to calculate working time out of a LOC/H value. The values of the real-world sample data are in a range from 0.48 minimum to 1618.52. If it is assumed, that the taken data are representative for any larger software organization some of the values can be eliminated instantly. As the assumption is, that commits are at least daily and outliers should not affect the LOC/H value, all rows with '$Percentage = 100\%$' and in the '*original*'-Section of the table can be eliminated.

Moreover, it can be seen, that depending on the way of calculation, the median and average of each possibility can deviate largely. Looking at the plots of the originating lines (Figures 3.7 and 3.11) one reason of the differentiation is clear: Spikes in the lines do influence the average but not less the median. Moreover, are most of the commits having a low number of LOC and less are larger ones. Therefore, the median values are typically lower than those of the average.

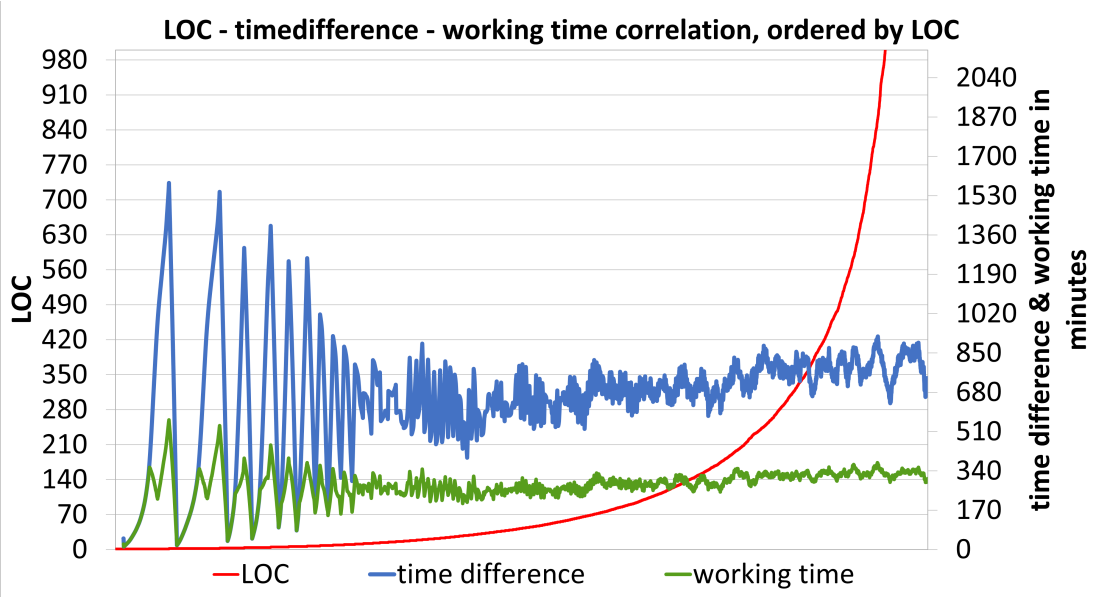
3.4.4 Individual LOC/H value

As a fourth option of how to handle individual commits, further analysis was conducted. In this section, the correlation between time difference and LOC will be analyzed more deeply. This idea is originating out of the previous analysis results, where it was shown, that the average and median LOC/H are differentiating. Therefore, the question to be answered now is:

How do the time differences, working time and number of LOC correlate?

To analyze this, the commits with its representative time differences (just considering commits within one workday, <36 Hours) and working time (calculated with the formula developed in Chapter 3.3.2.2) were ordered ascending by LOC as first criteria and then by time difference as second criteria. The result is shown in Figure 3.14.

Figure 3.14: LOC, time difference and working time correlation

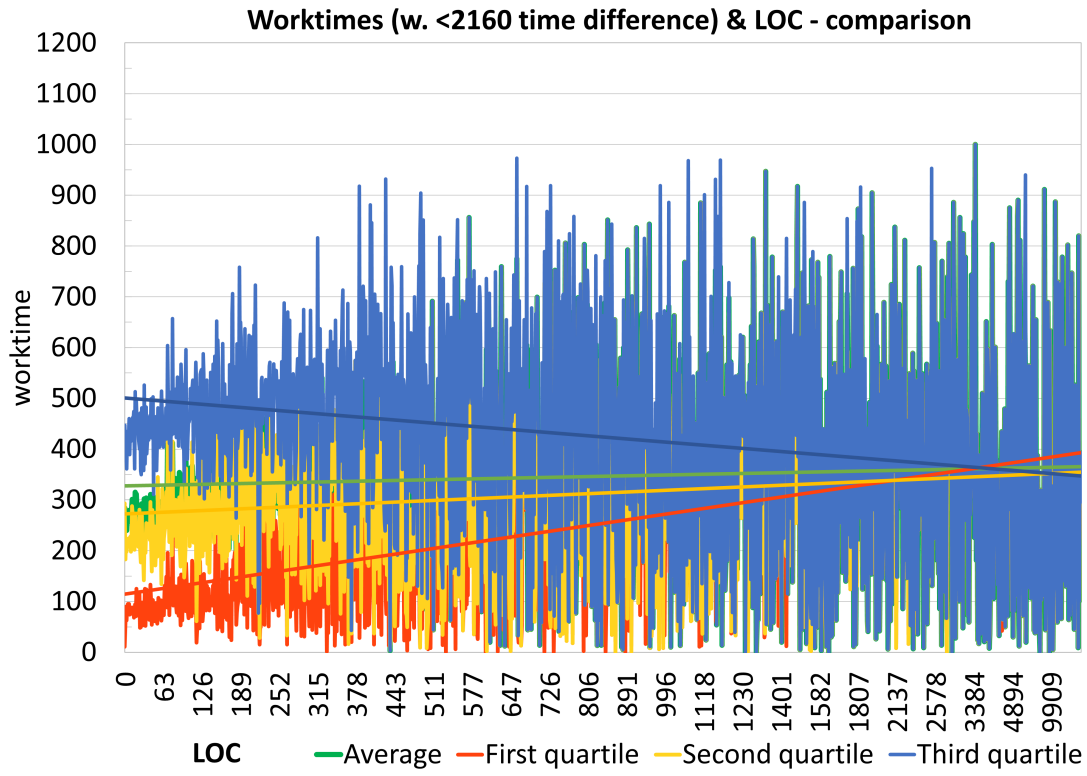


The plot in the figure shows on the exponentially rising number of LOC (left Y-Axis) and the correlating, fluctuating time differences and working time (right Y-Axis). Putting LOC, time differences and working time together several things can be recognized: On the one hand, the time differences and working times are high fluctuating, especially for smaller LOC. Reason is, that there are numerous small commits. Moreover, these large amounts of small commits are covering a large area of time differences. The larger the LOC committed are, the less commits are made and consequently the less fluctuation within the time differences and working time is. On the other hand, the difference between working time and time difference can clearly be seen for each LOC. It has to be noted, that the working time line is flatter than the line of time differences. Reason are, that time differences are not looking at e.g. night times like the work time calculation shown above does.
Additionally, despite the exponential rise of LOC, the time differences and working time are relatively steady and not also exponentially rising. Therefore, it can be concluded, that the working time is correlating with the LOC committed, but not in a way, that the two numbers are linked together one to one. The working time is indeed increasing (linear) with more LOC provided, but not exponentially in the same factor.

Figure 3.14 shows the correlations for all time differences. However, the same pattern and correlations were recognized for doing this analysis for different windows of time differences(e.g all time differences <3/6 hours, 3-15 hours, >12 hours). Even though the curves are sometimes flatter or steeper, the basic idea (No one to one correlation between working time and LOC) stays the same.

As this analysis showed that there is a huge deviation how the time differences are distributed, the next step was to show the working time and LOC correlation again. This time the plot is not ordered ascending by LOC but using the LOC as the X-axis and additionally showing the quartiles and average of the working time. The result is shown in Figure 3.15.

Figure 3.15: LOC and work time(quartiles, avg) comparison



The results are showing, how the working time for each LOC is distributed. In this figure it is more clearly (as already written above), that for low number of LOC committed, the difference between the working times is high (high fluctuation). There are many small commits, which results in a low first and high third quartile. As the number of commits per LOC get lower for higher LOC, the fluctuation range is smaller, which results into an almost similar plot at the right end of the scale. Additionally, calculating a linear regression line makes the results more obvious: At the left end of the plot, the regression lines are larger apart and approach each other as the number of LOC are rising (In fact, the quartiles are resulting in the same point for the highest LOC. Due to the nature of the regression calculation, they are crossing each other at some point and not directly ending at the right end of the axis)

Looking at the correlation between working time and LOC it is clear, that there is

no such element as an uniquely applicable value how the LOC and working time are correlating. For each LOC committed, a large variety of time difference can be observed. Additionally, for every time difference of a commit, a large variety of LOC can be observed from historic data.
Therefore, it can be concluded, that it is not possible to assign an exact working time to a commit by just having the LOC and time difference available.

A last option in this thesis, how (independent) commits can be assigned with working time is to choose one of the regression lines as calculation basis. As the second quartile and average liens are closely related to each other, this choice might be viable option. However, it has still to be decided by the use case and overall circumstanced if e.g. large commits (in this analysis: largest 5%) are also valued by this regression line, as the regression line is rising with larger commits, which might lead to a comparable lower working time per LOC than looking at smaller commits. Especially for large imports of other projects must be considered, if they were not already included at another point.

3.5 Concept Summary

Before an example implementation can be done, the overall concept will now be summarized again.

Regular commits within one workday(<2160 minutes time difference):

The concept shows the result after the four iterations of developing a working time concept. In the fourth iteration, several formulas (respectively algorithms) were introduced, on which basis the implementation can be done.

The Formula 3.5 puts together all the single equations shown in Chapter 3.3.2.2.

$$worktime = \begin{cases} \sum_{k=timestamp_{currentCommit}}^{timestamp_{previousCommit}} 1, & \text{if } timedifference < 360minutes, \\ worktime_{nightshare}, & \text{if } timedifference \geq 360 \text{ and } \leq 720minutes, \\ worktime_{proportional}, & \text{if } timedifference > 720 \text{ and } \leq 2160minutes \end{cases}$$

with:

$$worktime_{proportional} = \frac{\sum_{i=timestamp_{currentCommit}}^{timestamp_{previousCommit}} \text{numberOfCommits}_i}{\text{numberOfCommits}_{overall}} * \text{worktime}_{day}$$

$$worktime_{nightshare} = nightshare * worktime_{proportional} + (1 - nightshare) * daycoverage$$

$$nightshare = \frac{\sum_{i=timestamp_{currentCommit}}^{timestamp_{previousCommit}} x_i * 1}{\sum_{i=timestamp_{nightBegin}}^{timestamp_{nightEnd}} 1}$$

$$daycoverage = \sum_{j=timestamp_{currentCommit}}^{timestamp_{previousCommit}} x_j$$

$$worktime_{day} = \text{typical workday in minutes}$$

$$x_i = \begin{cases} 1, & \text{if } i \in nighttime, \\ 0, & \text{otherwise} \end{cases} \quad x_j = \begin{cases} 1, & \text{if } j \in daytime, \\ 0, & \text{otherwise} \end{cases} \tag{3.5}$$

As a short reminder, the following aspects must be given to implement the equations:

- The time difference from each commit to the previous one
- Time length of a typical workday in minutes ($worktime_{day}$)
- A limit of the maximum number of commits per timestamp to calculate the *nighttime*/*daytime* (*nightEnd* and *nightBegin* respectively) (See Figure 3.13) OR
- Manually set a *nighttime* and *daytime*

Irregular commits without time difference and commits >36 Hours apart:

All commits don't have any previous commit, are >2160 minutes apart or are other special cases (e.g. imported projects that needed to be evaluated again) can be assigned with a working time by using the LOC and working time correlation found out in Chapter 3.4.4. For this, either the average or median regression line is used to calculate the working time. To calculate the regression line, each working time with time difference <2160 minutes already must have been calculated. These commits are then grouped by LOC. For each LOC, the median or average working time is given. This array of values is then used for the linear regression. The median linear regression line for the sample dataset is:

$$worktime_{linear} = 0.0525280 * loc + 272.4397728 \tag{3.6}$$

Using the linear regression enables irregular commits to get work time assigned in height of the median work time provided by the other more exact commits.

4 Architecture, design, and implementation

In this Chapter a prototype implementation of the previous developed concept will be presented. At first, an architectural overview will be given (Chapter 4.1). Based on this architecture, the way information is flowing through the system will be explained(Chapter 4.2). After a brief overview, some technical details (Chapter 4.3) the work time calculation(Chapter 4.4) and cost calculation is shown (Chapter 4.5).

4.1 Architectural overview

The basic architecture of the system can logically be split into four parts, which are shown in Figure 4.1.

Part A (Top of the figure) represents the data sources used in the system. There are two main types of data sources needed to implement the transfer price. On the one side there are the commit data originating from a version control system. Depending on the type of version control system, the commits might either be directly transferred via an interface or (like in this thesis) be exported as CSV data and used in this format. On the other side there are also the cost data needed for the Cost Plus calculation, especially from a cost centre point of view. For a non-prototype implementation, the cost data should be directly imported from the accounting software.

Part B of the System shows the processing part of the Implementation. In this part, the algorithm is implemented (work time is calculated, accumulation of work time per project) and the Cost Plus Method is applied.

In Part C of the implementation, the costs per commit/transaction is then processed again so that the right data are outputted and the data format is suitable. Depending on the further usage (Part D), the data might be grouped by Project (Transferred costs per Project), by Organization (Transferred costs per organization) or by user (Individual user contributions, if needed).

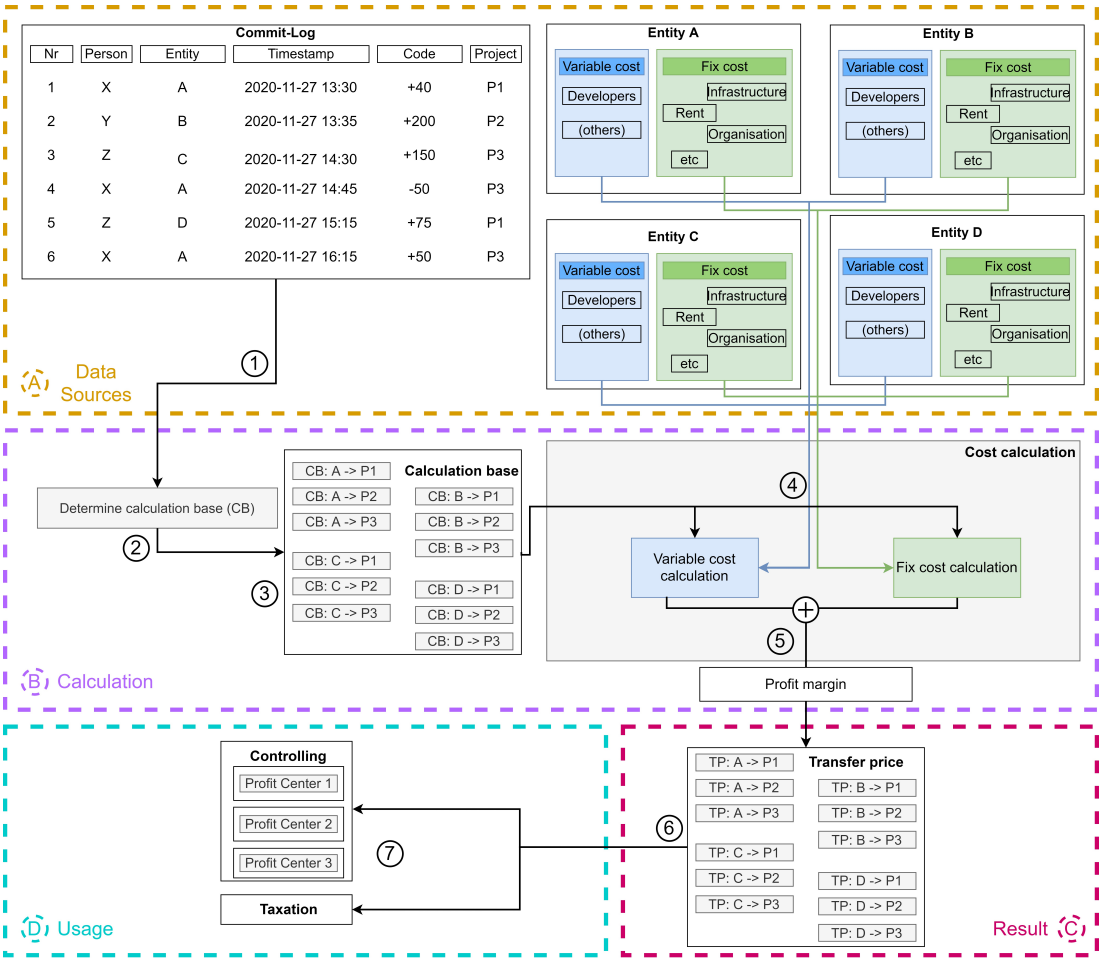


Figure 4.1: Architectural overview over the implemented System

4.2 Information flow

The processing done during the runtime of the program is also shown in Figure 4.1. The procedure is in alignment with the structure of the system previously explained.

Step 1 is the extraction and processing of the commit data. In this prototype implementation, the commits were preprocessed, loaded into the database and then are requested on demand.
The next step (Step 2) is then to calculate the work time for each commit according to the algorithm developed in this thesis (See Chapter 3.5). The result is again a list of commits, but this time with work times assigned to it.
Step 3 (when requested) is the aggregation of the work times from an user level to an organizational level to hide individual programmers performance.
Step 4 is loading the cost centre data from the database needed for the level

requested.
The information form Step 3 and 4 are consequently used to do the cost calculation for each organization. The result is list of costs, each committing entity is producing to contribute to certain software modules, which are consequently owned by others or the same entity again.
After adding the profit margin for each transaction (Step 5), the result is prepared according to the query(Step 6).
Lastly the results will be output (here: REST API) so that the results can be reused for different purposes.

4.3 Technical overview

Before the process is described in detail, a brief overview of the techniques used in the system will be given.

In Part A of the prototype system (See Figure 4.1) one PostgreSQL database was used, to which the commit data were imported. The commit data were exported from an enterprise version control system in CSV format.
The cost centre data were also stored in the database, as no dedicated connection to an external accounting tool was available.

Beside SQL for database queries, PHP was used as main language. The main reason in this thesis to use PHP as language was, that it was easy to implement a CSV-Import webpage for easy usage, which later might be extended to also show the work times, costs and transfer prices between the organizational units. To have a consistent choice of technology within one thesis, the processing (Part B) and output (REST API, Part D) was also done with PHP. The execution was done by an Apache Webserver.

Querying the data can be done with a REST Interface solely implemented with PHP (without extra packages). Implementing a REST API was chosen because, on the one hand, it is a modern and widely used approach (Neumann et al., 2018) and, on the other hand, enables the results of the implementation to be reused easier in further research. The data returned by the REST API are formatted in the JSON format.

4.4 Preparation and work time calculation

As a first step, the available commit data in CSV format were preprocessed and uploaded to the database. In this step, at first, a simple PHP Upload webpage was built. This form can be used to upload the CSV commit data. During the upload process, invalid commits were sorted out. As the sample data are only a part of a larger commit set (The commits belong to a platform as part of a

larger product portfolio), all commits where the author is belonging to a willingly unsupervised organization were not uploaded. Additionally, commits to modules not included in the platform were also ignored.

When requesting the data for usage with a SQL-Query, the commits were not simply returned, but also further preprocessed. As in the list of commits each entry represents one single file change, the commits were aggregated, so that for one upload (also called Patch (Capraro, 2020)) is calculated one time. In this step, the number of added/deleted/modified lines were also summed up.
In addition, the time differences for each commit to the previous commit of the same user were calculated.

After returning the commit data with time differences from the database, the calculation of the algorithm presented in Chapter 3.5 can be done.
First, all commits with time differences <2160 minutes are calculated with the Formula 3.5. After this is done, the regression lines are calculated. To do this, all commits including their work time are sorted by LOC ascending as first criteria. For each LOC, the average or 2nd quartile work time must be calculated (As they are similar, this thesis uses the 2nd quartile data). The 2nd Quartile work time per LOC data is then used for the regression calculation. The result (like equation 3.6) is used for all other commits, which are those >2160 minutes time difference and those without time difference available.

The final result after this step is a complete list of commits with their estimated work time, which can now be used to calculate costs with the Cost Plus Method.

4.5 Cost calculation

The calculation of the transfer price is the next. Before the calculation itself can be done, some preparations must be made, which will be explained in this chapter.

4.5.1 Cost structure

In this thesis, the work time concept is in focus, while the implementation itself is a prototype. Therefore, having a real-world cost structure imported from an accounting system was not the main task.

The costs used for calculation where randomly created as the height of the costs plays only a secondary role and the procedure of calculation and structure of the costs is more important. For every transaction made between entities, the direct and indirect costs (See Chapter 2.1.4) must be known. All direct costs (e. g. dedicated Server in Entity A for a software project which belonging to Entity B) can be assigned directly to the transaction.

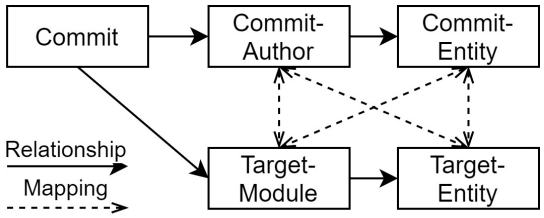
All indirect costs (e. g. personnel costs) must be available for each cost centre (Organizational unit). These costs can later then split according to the workload information provided by the commits. In this thesis, the personnel costs were calculated by taking the number of developers of each organizational unit and multiplying them with a randomly distributed salary for each employee. Additionally, four other indirect costs were generated, which represent other indirect costs like rent and maintenance. In addition, the costs were differentiated by year, so that each fiscal year can be treated individually, as it is usual in taxation.

The sample data are giving organizational data as granular as the scrum teams within each organization. However, the costs used here are generated not on scrum-team level, but on the organizational level, the scrum teams are belonging to.

4.5.2 Calculate cost split

At this point of the calculation process, the cost calculation itself is prepared by processing the list of commit data with its work times. In addition, the organizational hierarchy, number of people for each entity and a user (author of a commit) to entity mapping is required. In addition it must be known, which software project (or module) belongs to which organizational unit.

Figure 4.2: Commit, Author and module dependencies



In Figure 4.2 the different attributes of a commit and how they are connected to each other is shown. As part of the commit, the data of the committing author and the target module is directly known. Moreover, it is also known, which entity a user or a software module belongs to. Therefore, the entity which receives the work (Target-Entity) and the entity which contributes (Commit-Entity) can be found out.

On basis of these relationships, different mappings can be calculated. Based on the request of the user it can, for example, be calculated how much a committing entity contributes to projects belonging to other entities or how much workflows to which projects. The mappings can also be done in the other direction.

Depending on the detailed request, the work time of the commit list will now be aggregated according to the structure needed. As this sample implementation

is about finding transfer prices for taxation, the goal will be to find out, how much a Commit-Entity is contributing to other entities (crossing tax boundaries). Therefore, the organizational structure is the basis. For each organizational unit, the number of minutes worked for each Target-Entity will be summed up. The result is a list of organizational units with the work time contributing to each other unit.

Having the work time for each Commit-Entity to Target-Entity mapping, a share can be calculated, how much percent of the work time was spent for which project. This is done, because for the cost calculation it must only be known, how the indirect costs must be split (See Chapter 2.1.4).

Depending on the hierarchy requested, the work times of lower-level organizations may be aggregated, to fit the level needed. Moreover, the overall annual work time (e. g. manual guess or imported by personnel system) and annual work time of analyzed developers must be known.

4.5.3 Conduct cost calculation

The final step is the cost calculation itself. Based on the list of organizations, their work time and percentage share, the costs can now easy be calculated.

At this point, it will be (like in Chapter 2.1.4 explained), differentiated again between direct and indirect costs. All direct costs can directly be assigned to one transaction. In our case, this may for example be a dedicated Server Entity A uses to be able to contribute to a module owned by Entity B.
All indirect costs are structured by cost centre (organizational unit), as the entities are those who contribute. Therefore, the costs of the current organization and year must be requested. In the preparation, the work time shares for each organization where calculated. Based on this share, the costs per entity (cost centre) can be split. If, for example, Entity A contributes 60% of the work time for its own projects, 30% of the work time for Entity B and 10% of the work time for Entity C, the costs can/must be split according to that share. This affects all costs of the cost centre, which cannot directly be assigned to one project (like personnel costs, rent etc.).

At this point the difference of calculating Top-Down (explained here) and validating Bottom-up (see Chapter 3.2.1) can be emphasized again. At this point the organization must decide depending on the use-case, if all costs of an organization are included for the cost calculation (all commits are measured) or if just a part of the overall number of commits are being analyzed and therefore just a smaller part of the costs must be split according to the shares. Moreover, it would be possible to validate if the guessed cost share/work time for the observed projects ware concluded in the right way (Bottom-Up validation).

The last step is adding a profit margin for each transaction (5% flat in our prototype). After this is done, the result and consequently the transfer price is calculated. In our prototype implementation, the result is a list of organizations, including the work time spend, costs, profit margin and overall transfer price for each entity the contributed projects are belonging to.

This list may now be used again for further processing in taxation, accounting, or other software. As already said, the prototype implementation makes it also easy not only to structure by organization, but also to get e.g. the Commit-Author to Target-Entity mapping. This information can now be accessed over the REST API which was implemented.

The endpoints of the REST API are in Appendix A. A demonstration (Sample output) of the API is shown in Appendix B.

5 Evaluation

The evaluation is no central part of this thesis but should be conducted in further research. Therefore, this chapter will show rather how the evaluation can be done for this work.

The main problem, this thesis solved was to calculate work times from a commit log using statistical analysis. The sample use case the calculation was developed for is cost calculation. Therefore, the evaluation should also target to evaluate the cost share in the first place.

One simple evaluation possibility is to conduct interviews with the data owner showing the results of the algorithm. To do this, people with a good overview over the development activities must be questioned. Ideally, the algorithm might be applied to more recent development activities in one sample organizational unit again and then evaluated with the manager of the department or a person knowledgeable over the internal work distribution in that business entity.

A second way of evaluation is to take the algorithm and execute it at historic data where transfer prices are known. In this way, the results can be directly compared to real world data. However, it is likely that (as the algorithm in only an estimation) the results of the algorithm and real-world data are differentiating. For that reason, additionally an expert in transfer pricing might be conducted to compare and rate the result.

A third way for evaluation would be to systematically write down or track work times at the developers workplace. To do that, either the detailed work time per commit must be measured (e. g through tracking file changes) or just the overall work time per day in combination with the organizational overhead (timestamps and which project) is needed. In the latter case, the overhead helps to understand, how the work time not dedicated for development is distributed over the day. The algorithm presented in this thesis includes overhead in the calculation. Using exact work times therefore not only helps to improve the cost share, but also might enable to use detailed work time calculation for a large variate of further uses cases.

6 Conclusion

6.1 Summary

The goal of this thesis was to develop a concept how transfer prices, especially Cost Plus, can be calculated for software development by using a list of code contributions.
The solution was created iterative through analyzing the time difference between commits, the timestamp of the commit and the relationship between LOC and calculated work time. The extracted dependencies were used to develop an algorithm which can be used to assign every commit an estimation of work time spend on it.

In addition to the conceptual work, a prototype implementation was done, where not only the work times where calculated, but also used for a sample transfer price calculation. This prototype makes it easy to reuse work times, costs, transfer prices and different sorts of workflows between commits and modules in future research.

6.2 Limitations and outlook

When applying the developed algorithm, it is worth to keep in mind, what it was designed for and for what not. The main goal of this algorithm is to calculate work times usable for cost calculation. Therefore, the algorithm was designed to output a work distribution (share per project) for each organization, even though single commits might be inaccurate. With that context in mind, some limitations are important to know, but which also show potential for future research.

One limitation is, that the algorithm was in this thesis especially design to be applicable for the Cost-Plus method in taxation. Future work might show if the concept is transferable (especially from a tax point of view) to other transfer pricing method and which information are additionally needed for those methods. Moreover, it might also be worth analyzing, how the work time calculation is helpful for controlling and management to lower the boundaries of introducing

and performing inner source software development.

From an technical point of view future work must also show, how exact work time of single commits really are. This can be done by calculating the deviation to real work time per commit and improve the algorithm with that data.

Even though the algorithm developed in this thesis is not yet uniquely applicable for all use cases, it might set the basis to improve software development, taxation, controlling and ultimately strategic management decisions based on exactly measured work flows between and within organizations.

Appendices

A REST API overview

Method	Endpoint	Description
GET	/commit/rank/:rank	Returns the list of commits, grouped by commit-date per author, with rank, extracted hour and time difference to previous commit
GET	/commit/numberPerHour/	Returns a list of commit times and number of commits made in this time
GET	/commit/hourPriceComplex/	Returns a list of commits with time differences, loc/h performance value and work time calculated by the (internally called) complex hour price method (Working time concept from fourth iteration, all commits with time difference 2160 minutes, Chapter 3.3.2.2)
GET	/commit/workingTime/	Returns a list of commits with the working time assigned. All commits with an previous commit by the same user are calculated by the hourPriceComplex-Formula (360/360-720/720-2160 difference). All irregular commits are calculated by using the LOC-Worktime relationship (median regression line). See Chapter 3.5

GET	/nightTime/	Returns the start of the night(dayEndTime) and begin of the day(dayStartTime). If no commits are found, it returns default time for the night
PUT	/commit/updateHourPriceComplex/	Updates intermediate used table for linear regression calculation with data from regular commits
GET	/commit/wtPerLocData/	Returns the avg and quartiles (q1_median_wt, q2_median_wt, q3_median_wt) for each LOC found in the dataset. See Figure 3.15
GET	/commit/wtPerLocRegLines/	Returns the regression lines for the avg and quartiles (q1_median_wt, q2_median_wt, q3_median_wt). Basic data is working_time foreach LOC found in the dataset. See Figure 3.15
GET	/costs/transferPrice/	Returns for each org unit the transfer prices and minutes worked at projects from other organizations

Table A.1: Overview of REST endpoints

B Example API outputs

Case	JSON Output
Regular commit with time difference < 360 min	{"module": "Module A", "commit_date": "2015-01-02 15:18:02", "owner": "Person A", "time_difference": "23", "loc": "462", "working_time": "23"}
Regular commit with time difference 360-720 min	{"module": "Module B", "commit_date": "2016-02-15 09:35:27", "owner": "Person B", "time_difference": "655", "loc": "70", "night_share": 1, "working_time": 50}
Regular commit with time difference 720-2160 min	{"module": "Module C", "commit_date": "2016-04-08 11:07:01", "owner": "Person C", "time_difference": "953", "loc": "684", "working_time": 154}
Irregular commit with no time difference	{"module": "Module D", "commit_date": "2015-01-06 17:58:19", "owner": "Person D", "time_difference": null, "loc": "227", "working_time": 268}

Example transfer price calculation	{"org_name": "Organisation-A-A-A",
	"parent_org": "Organisation-A-A",
	"overall_personnel_sum": 16,
	"transfer_data": {
	"2015":
	{
	"Organisation-A-A": {
	"share": 84.96,
	"costs": 47701.28,
	"transfer_price": 50086.34
	},
	"Organisation-A-A-A": {
	"share": 7.31,
	"costs": 4106.04,
	"transfer_price": 4311.34
	},
	"Organisation-B-A": {
	"share": 6.53,
	"costs": 3669.06,
	"transfer_price": 3852.51
	},
	"Organisation-C-A": {
	"share": 0.80
	"costs": 448.18,
	"transfer_price": 470.59
	},
	"Organisation-D-A": {
	"share": 0.40,
	"costs": 223.23,
	"transfer_price": 234.39
	}}}}

Table B.1: Example API output

References

Capraro, M. (2020). *Measuring inner source collaboration* (doctoralthesis). Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU).

Carroll, N., Morgan, L. & Conboy, K. (2018). Examining the impact of adopting inner source software practices. *Proceedings of the 14th International Symposium on Open Collaboration*. https://doi.org/10.1145/3233391.3233530

Cooper, D. & Stol, K.-J. (2018). *Adopting innersource: Principles and case studies*.

Feller, J. & Fitzgerald, B. (2000). A framework analysis of the open source software development paradigm, 58–69.

Fluri, J. & Deck, K.-G. (2018). Automatisierte Kollaboration und Prozesse in der Softwareentwicklung - Wandel von Unternehmenskultur und Unternehmensstruktur. In K. O. Tokarski, J. Schellinger & P. Berchtold (Eds.), *Strategische Organisation: Aktuelle Grundfragen der Organisationsgestaltung* (pp. 259–283). Springer Fachmedien Wiesbaden. https://doi.org/10.1007/978-3-658-18246-5_12

Fuller, R. (2019). Functional organization of software groups considered harmful. *2019 IEEE/ACM International Conference on Software and System Processes (ICSSP)*, 120–124. https://doi.org/10.1109/ICSSP.2019.00024

Hanken, J., Kleinhietpaß, G. & Lagarden, M. (2017). *Verrechnungspreise. Praxisleitfaden für Controller und Steuerexperten* (2. Auflage). Haufe Gruppe.

Krause, S. & Pellens, B. (Eds.). (2018). Herausforderungen neuer digitaler Geschäftsmodelle für die Bestimmung von Verrechnungspreisen. In *Betriebswirtschaftliche Implikationen der digitalen Transformation* (pp. 143–165). Springer Fachmedien Wiesbaden. https://doi.org/10.1007/978-3-658-18751-4_8

Leite, L., Kon, F., Pinto, G. & Meirelles, P. (2020). Platform teams: An organizational structure for continuous delivery. *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 505–511. https://doi.org/10.1145/3387940.3391455

Neumann, A., Laranjeiro, N. & Bernardino, J. (2018). An analysis of public rest web service apis. *IEEE Transactions on Services Computing*, *PP*, 1–1. https://doi.org/10.1109/TSC.2018.2847344

OECD. (2017a). *Gewährleistung der Übereinstimmung zwischen Verrechnungspreisergebnissen und Wertschöpfung*. https://doi.org/10.1787/9789264274297-de

OECD. (2017b). *Oecd transfer pricing guidelines for multinational enterprises and tax administrations 2017*. https://doi.org/10.1787/tpg-2017-en

Olbert, M. & Spengel, C. (2017). International taxation in the digital economy : Challenge accepted? *World Tax Journal : WTJ*, *9*(1), 3–46. https://madoc.bib.uni-mannheim.de/41867/

Open Source Initiative. (2007). *The open source definition*. Retrieved March 11, 2021, from https://opensource.org/osd

Prangenberg, A., Stahl, M. & Topp, J. (2011). *Verrechnungspreise in konzernen*. Hans-Böckler-Stiftung.

Riehle, D., Capraro, M., Kips, D. & Horn, L. (2016). Inner source in platform-based product engineering. *IEEE Transactions on Software Engineering*, *42*, 1162–1177. https://doi.org/10.1109/TSE.2016.2554553

Schwerdt, D. (2016). Verrechnungspreismethoden und Ökonomische Analyse. In R. Dawid (Ed.), *Verrechnungspreise: Grundlagen und Praxis* (pp. 163–241). Springer Fachmedien Wiesbaden. https://doi.org/10.1007/978-3-658-09377-8_5

Stol, K.-J., Avgeriou, P., Babar, M. A., Lucas, Y. & Fitzgerald, B. (2014). Key factors for adopting inner source. *ACM Trans. Softw. Eng. Methodol.*, *23*(2). https://doi.org/10.1145/2533685

Stol, K.-J. & Fitzgerald, B. (2015). Inner source–adopting open source development practices in organizations: A tutorial. *IEEE Software*, *32*(4), 60–67. https://doi.org/10.1109/MS.2014.77

United Nations. (2014). *United nations practical manual on transfer pricing for developing countries*. United Nations. https://www.un-ilibrary.org/content/books/9789210561372

Weber, W., Kabst, R. & Baum, M. (2014). *Einführung in die Betriebswirtschaftslehre*. Gabler Verlag. https://doi.org/10.1007/978-3-8349-4677-5_1