A FRAMEWORK FOR OBTAINING AND PREDICTING SOFTWARE MAINTENANCE EFFORT ESTIMATORS

CARLOS VINICIUS ANDRADE SILVA

A Combined Qualitative and Quantitative Study in Open Source Software
2013 – September
"Success is going from failure to failure without losing enthusiasm."
— Winston Churchill

Traue keiner Statistik, die du nicht selber gefälscht hast.

Dedicated to the loving memory of my grandparents.
ABSTRACT

Background: Current state of the art work in maintenance effort prediction are made in a coarse grained level (e.g. project level). Although such models are useful for predicting new similar software project expected effort, they can’t be used by software architects whose interest focus in a per file per release grained level for decision making. More generally, a fine grained level such as file can also be used in order to assess technical debt. Furthermore, higher granularity prediction models usually demand effort in clock time, which is rarely available both in open source and industry environments.

Methods: We created and applied in Open Source Software (OSS) a framework that is able to empirically acquire maintenance effort estimators and further use them as predictions when effort in clock hours is not available. To illustrate the framework, we apply it in open source collected data where effort in time is not available.

The framework consists of a combination of qualitative and quantitative methods and is divided in three phases. The first consists of obtaining proxy effort estimators and file metrics; the second consists of assembling data that contains the chosen estimators and file metrics; the last consists of predicting proxy effort estimators from structural complexity metrics.

Results: We empirically obtained a rich and descriptive abstraction from the main concern and how it is processed in the analyzed open source projects, from which we derived as maintenance effort estimators churn (the amount of lines added and removed from a file), actions (the amount of times a file is changed in order to address an issue), and discussion (the amount of comments associated to an issue in an issue tracker), the later being the most representative effort estimator among the three. We observed that churn and discussion were correlated consistently in all project releases, followed by actions and churn. While having low spearman correlation with the obtained effort cost estimators, we were able to successfully create prediction models from file metrics, actions and churn (which are obtained from repository) in order to predict discussion (which is obtained from issue trackers).

Conclusions: Given that from previous work churn has been reported as correlated and able to predict effort in clock-time, from our open source case study we conclude that churn alone may be insufficient to account for all kinds of effort in a given domain, further strengthen the motivation of our proposed framework.
RESUMO

Contexto: O atual estado da arte em predição de esforço de manutenção tem sido realizado em uma granularidade alta (e.g. nível de projeto). Apesar que estes modelos são úteis em predizer esforço de novos projetos similares, eles não podem ser usados por arquitetos de software cujo interesse está a nível de granularidade arquivo por release para tomada de decisões. De modo geral, uma granularidade mais fina como a de arquivo também pode ser usada de modo a avaliar débito técnico. Além disso, uma granularidade mais alta para modelos de predição normalmente demandam esforço medido em horas, o que raramente se encontra disponível tanto em projetos open source como em ambientes industriais.

Métodos: Nós criamos e aplicamos em OSS um framework que é capaz de obter estimadores de esforço de manutenção quando esforço em horas não se encontra disponível. Para ilustrar o framework, nós aplicamos ele em dados minerados de projetos open source onde tempo não não se encontrava disponível. O framework consiste de uma combinação de métodos qualitativos e quantitativos, e é dividido em três fases. A primeira consiste em obter estimadores de esforço indiretos e métricas de código; a segunda consiste em organizar os dados que contem os estimadores escolhidos e as métricas de código; o último consiste em realizar predições dos estimadores indiretos de esforço a partir das métricas de complexidade estrutural.

Resultados: Nós empiricamente obtemos uma descrição rica e descriptiva da principal preocupação e como ela é processada nos projetos open source analisados, de onde nós derivamos como estimadores de esforço de manutenção churn (a quantidade de linhas de código adicionadas e removidas de um arquivo), ações (a quantidade de vezes que um arquivo é modificado para resolver um issue num issue tracker), e discussion (a quantidade de comentários associados a cada issue em um issue tracker), o último sendo o mais representativo estimador de esforço dentre os três. Nós observamos que churn e discussion estavam correlacionados de modo consistente em todos os releases de cada projeto, seuidos de actions e churn. Enquanto possuindo um valor baixo de correlação de spearman, nós fomos capazes de criar com sucesso modelos de predição de métricas de código, actions e churn (que são obtidos de um repositório), de modo a predizer discussion (que é obtido de issue trackers).

Conclusões: Dado que de trabalhos passados churn já foi mencionado como correlacionado e capaz de predizer esforço em horas, de nosso estudo de caso de open source nós concluímos que churn apenas pode não ser o suficiente para cobrir todos os tipos de esforço em um
dado domínio, e portanto fortalecendo a motivação de nosso framework proposto.
PUBLICATIONS

Some ideas and figures have appeared previously in the following publications:

Only a few find the way, some don’t recognize it
when they do - some... don’t ever want to.
— The Cheshire Cat - Aline in Wonderland - Lewis Carroll

ACKNOWLEDGMENTS

I would like first and foremost to thank my mother and father, for which had not invested so much in my education, supported me all along and understood my absence in social life in pursuing my academic endeavors, I would not have survived one step ahead.

I am in eternal debt to Eduardo Almeida, who saw in me a researcher in my first year in school and taught me many many things to be one. To Ivan Machado, who constantly gave me advice in everything about academia whenever I needed more than a mentor or a friend could ever give. I am also in eternal debt to Rick Kazman who, like Eduardo, saw a researcher on me and did everything he could to make me grow even further as one. His good will, amazingly fast e-mail replies, patience and granted freedom gave me wings to fly even higher than I ever imagined. To Yuanfang Cai, Florian Uunk, Peppo Valetto, Lu Xiao, Noah Black, Ira Monarch, who were major collaborators in published and undergoing works, open-minded listeners to my ideas and very compromised people to their work. To all my institution professors of whom I am proud of for knowing to recognize not only right answers, but the effort their students put to achieve them.

To my very dear friend Ruby Romero, who I had the honor to be my room neighbor during my one year in United States and still stay in touch ever since. To all my second brazilian family of the Science Without Borders and Stevens Institute of Technology staff, who I do not dare to cite names as are all equally dear and important to me.

To my friends Clara Aderne, David Gorender, Gustavo Sampaio and Mariana Uaquim for also being so understanding in having a friend always busy and forcing me back to see the sunlight. To my old friends Daniel Almeida and Monique Barbosa to their constantly inspiring motivation on learning new things and pursue their dreams. To my far away friends Talita Galanti, Silvia Dominguez, and Mara Duarte for tagging along on nonsensical talks very late at night and keep me up while I was trying to get this done.

To my school friends of whom I as well do not dare to cite names, as are always together when I have the pleasure to meet, some from so many years since I entered school, and some who I not so long met.
To Misty Davies, for scouting me in NASA Ames, and accepting me to work under her wings as a summer intern, and still stay in touch.

To all of you, thank you.
CONTENTS

1 INTRODUCTION 1
  1.1 Context 1
  1.2 Introduction 2
  1.3 Research Method 5
    1.3.1 Extract Indirect Maintenance Effort Data 5
    1.3.2 Build and validate indirect maintenance effort models 6
    1.3.3 Predict indirect maintenance effort 6
    1.3.4 Open Source Software Case study 6

I QUANTIFYING MAINTENANCE EFFORT PROXIES 7

2 CLASSIC GROUNDED THEORY 9
  2.1 Introduction 9
  2.2 Classical Grounded Theory 11
    2.2.1 Grounded Theory Building Blocks 11
    2.2.2 Grounded Theory Phases 15
    2.2.3 Conclusions About the Method Interpretation 18
  2.3 Other Grounded Theory Work 18
  2.4 Dataset and Tools 21
  2.5 Results 24
    2.5.1 Chance Need 24
    2.5.2 Piping Problems 25
    2.5.3 Iterative Informed Consent (IIC) 26
    2.5.4 The IIC Marketplace 28
    2.5.5 Iteration Blockers 28
    2.5.6 Uncertainty of Conclusion 29
    2.5.7 Project Survival 30
    2.5.8 IIC as a Basic Social Process 30
  2.6 Evaluation 31
  2.7 Conclusions 33
  2.8 Chapter Summary 34

3 LITERATURE REVIEW 37
  3.0.1 Software Development Effort Estimation 39
  3.0.2 Dataset Concerns 40
  3.0.3 Software Maintenance Effort Estimation 41
  3.0.4 Effort Proxies 42
  3.0.5 File Metrics 43
  3.1 Chapter Summary 43

II DATA COLLECTION AND DATA STORAGE 45

4 SOFTWARE ENGINEERING DATASET REVIEW 47
  4.1 Introduction 47
  4.2 Raw Data 47
4.3 Threats to Validity  52
  4.3.1 Mining Threats to Validity  52
  4.3.2 Inherited Threats  53
4.4 Chapter Summary  55

III DATA ANALYSIS  57
5 PREDICTING EFFORT  59
  5.1 Pre-processing  59
  5.2 Exploratory Analysis  60
  5.3 Correlation Analysis  63
  5.4 Prediction Analysis  69
    5.4.1 Support Vector Machines  72
    5.4.2 Artificial Neural Networks  74
    5.4.3 Ensemble Methods  74
    5.4.4 Non-interval prediction  74
    5.4.5 Other models  76
  5.5 Chapter Summary  77

IV CONCLUSION AND FUTURE WORK  79
6 CONCLUDING REMARKS  81
  6.1 Conclusions  81
  6.2 Limitations  81
  6.3 Future Work  82

BIBLIOGRAPHY  85

V APPENDIX  99
A CLASSIC GROUNDED THEORY MEMOS  101
  A.1 Memos generated from Data  101
  A.2 Memos of Memos  106
  A.3 Sorted Memos  108
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Framework for obtaining and predicting maintenance effort estimators</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Alternative Framework for obtaining and predicting maintenance effort estimators</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Iteration Blocker Branch. Lack of time is one of the properties of iteration blocker. We dispose the codes over a mind map to organize the codes. The core theory (not shown) remains as the root on its final version, leading to different branches (related categories), one of which is Iteration Blocker, having as one of its properties the Lack of Time. Text information that generated the property are exposed as leaves.</td>
</tr>
<tr>
<td>Figure 4</td>
<td>A coded JIRA issue report. In a word document, comments were used for memoing and coding. Memoing and coding on each document was distinguished by using the prefix 'M:' and 'C:'. This made it easier to query the memos after they were moved to mind maps, for organizing them to the memo sorting step</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Categories and Relationships in Jakarta</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Phenomena to Data workflow</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Patch contribution workflow</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Commit submission workflow</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Discussion Histograms in Derby</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Discussion Histograms in Lucene</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Discussion Histograms in Pdfbox</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Discussion Histograms in Ivy</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Derby Correlation Plot. Cost Estimators versus cost estimators, and cost estimators versus file metrics pairs are computed</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Lucene Correlation Plot. Cost Estimators versus cost estimators, and cost estimators versus file metrics pairs are computed</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Pdfbox Correlation Plot. Cost Estimators versus cost estimators, and cost estimators versus file metrics pairs are computed</td>
</tr>
</tbody>
</table>
Figure 16  Ivy Correlation Plot. Cost Estimators versus cost estimators, and cost estimators versus file metrics pairs are computed. 72

LIST OF TABLES

Table 2  Partial data table containing all maintenance effort estimators and two of the file metrics of CKJM. File path, issue code, release number and file metrics are omitted due to space, but uniquely identify each row of this table. 51
Table 3  Derby 60
Table 4  Lucene 61
Table 5  Pdfbox 61
Table 6  Ivy 62
Table 7  Derby 62
Table 8  Lucene 62
Table 9  Pdfbox 62
Table 10  Ivy 63
Table 11  Derby Quartiles 73
Table 12  Pdfbox Quartiles 73
Table 13  Derby Support Vector Machine Confusion Matrix 73
Table 14  Pdfbox Support Vector Machine Confusion Matrix 74
Table 15  Derby Feed Forward Back Propagation Artificial Neural Network Confusion Matrix 74
Table 16  Pdfbox Feed Forward Back Propagation Artificial Neural Network Confusion Matrix 74
Table 17  Derby Random Forests Confusion Matrix 74
Table 18  Pdfbox Random Forests Confusion Matrix 75
Table 19  Derby AdaBoost M2 Confusion Matrix 75
Table 20  Pdfbox AdaBoost M2 Confusion Matrix 75
Table 21  Derby Bagging Confusion Matrix 75
LISTINGS

ACRONYMS

OSS  Open Source Software
GT   Grounded Theory
IIC  Iterative Informed Consent
TSE  Transactions on Software Engineering
TOSEM Transactions on Software Engineering and Methodology
SMR  Journal of Software: Evolution and Process
ICSM International Conference on Software Maintenance
CSMR Conference on Software Maintenance and Reengineering
PROMISE Predictor Models in Software Engineering
MSR  Mining Software Repositories
WCRE Working Conference on Reverse Engineering
ICSE International Conference on Software Engineering
ESEM Empirical Software Engineering and Measurement
SAC  Symposium on Applied Computing
OLS  Ordinary Least Square
INTRODUCTION

Precisely, Gentleman, the only way to achieve the impossible, is to believe it’s possible.

— Charles Kingsleigh - Aline in Wonderland - Lewis Carroll

Figure 1: Framework for obtaining and predicting maintenance effort estimators

1.1 CONTEXT

Estimating the duration, effort, cost and complexity of software development activities is of vital importance for IT management, but notoriously challenging to do well. Molukken and Jorgensen’s review[57] showed that 60-80% of all software projects run over budget by 33% on average. More than half of the total development effort in software projects is spent on the maintenance phase [48].

During the maintenance phase, software tends to age [63] and the code base gets cluttered by an accumulation of changes, often referred to as technical debt. Like a financial debt, the technical debt incurs interest payments, which come in the form of the extra effort that we have to do in future development because of the quick and dirty design choice. We can choose to continue paying the interest, or we can pay down the principal by refactoring the quick and dirty design into the better design. Although it costs to pay down the principal, we gain by reduced interest payments in the future 1. When technical debt is not addressed, further development will be hindered. One major way to address technical debt is code (re-)modularization through refactoring.

If decision-makers (e.g. project managers) do not have good insight into the benefits of refactoring, it is difficult to know when to refactor, if at all. Numerous prediction models have been proposed recently to identify components that are error-prone [58, 38, 28], or to predict

Many studies have been conducted in respect to estimating effort or cost estimation and also many systematic reviews. Developing new models is the biggest research topic in software effort estimation since 1980s. There is also not an agreement over the terminology of this project development cost and effort \[56, 34\]. But their existing work does not directly support a project’s decision-makers in answering the following question: \textit{when is it worthwhile to refactor the software to reduce the complexity and make it better modularized?}

While the costs of modularization activities such as refactoring are significant and immediate, their benefits are often largely invisible, intangible and long-term. It has been known for decades that modularity decay can cause substantial problems in projects, such as reduced ability to provide new functionality and fix bugs, operational failures, and, in the extreme, canceled projects. But there is no established quantitative association between modularity variation and maintenance effort variation. It means there is no way for a decision-maker to know, with confidence, if a project’s modularity gets worse (or better) how much more (or less) it will cost to maintain and extend. Without such a foundation, it is difficult to predict the costs of the technical debt incurred from a deterioration in a project’s modularity. And it is equally difficult for decision-makers to justify the potential cost-savings from a proposed refactoring activity.

In addition to non established quantification between file and effort, the availability of effort data in a file grain level is virtually non-existent. State-of-the-art related work to ours is by Shihab et al.\[71\], in which mined OSS effort data across four projects are only rarely logged across bug reports (0.02\%, 0.02\%, 0.019\%, 0.078\% in Hibernate, JBoss, Mule and Spring respectively).

Given such under representative effort data, a need for a framework that empirically generate non-time proxy maintenance effort estimators is needed so a quantified relationship between refactoring and effort can be achieved.

Our research aims to provide an empirical foundation upon which sound refactoring decisions may be based, by relating variation in the complexity of code to variation in effort—and hence cost. We do so by creating a framework that provide us proxy maintenance effort estimators, structural complexity file metrics and quantify their relationship.

Concretely, in order to achieve our research goal, we must therefore: (i) Establish proxy maintenance effort estimators (ch. 2), (ii) select a set of representative file metrics of structural complexity (ch. 3) and (iii) obtain and quantify their relationship (ch. 4, ch. 5) as indicated by figure 1.
field, despite efforts of generating taxonomies due to their subjectiveness [41].

We decided using as a starting point the definition of Wen et al. [78], who conducted recently (2012) a systematic review of machine learning algorithms for the purpose of effort estimation models: according to Wen et al., software effort estimation can be branched in both software development effort estimation and software maintenance effort estimation. Also, software cost estimation is often used interchangeably with software effort estimation, though effort just forms the major part of software cost.

Broadly, software maintenance, which is the process of enhancing and optimizing software, can be classified in software support, source code change, and documentation update.

Within software maintenance source code change can be classified as corrective, perfective, or adaptive. Corrective code change is semantic-preserving and performed in an artifact to remove residual fault. Both Perfective and adaptive are non-semantic preserving, and are performed in an software artifact to change functionality and/or software property (e.g. security, performance) [81].

In this work, we are mainly interested in maintenance effort models. Specifically, our analysis focus on software maintenance source code change. This is due to the flexibility of these models to account for much more finer grained kind of data for analysis. By being able to reason about effort in a file grain level, we are able to inquire ourselves if there could be a relation between effort and complexity code metrics. If this is true, we could be able to make much finer grained estimations in respect to maintenance effort. For example, we would be able to know how much effort is being performed in a given file or how much effort would it demand.

We chose OSS as a case study due to the rich diversity and access to a considerable amount of available data. Furthermore, from a practical need, other works have shown interest in predicting maintenance effort for OSS [77].

As results from applying GT, a method to qualitatively analyze textual information, to OSS logs, we defined as maintenance effort estimators in OSS discussion, churn and actions. Respectively, they reflect the amount of comment posts in an issue tracker, the amount of changed lines of code in a file, and the amount of times a given file was changed to address an issue. For structural complexity metrics, we used the C&K metrics suite, as defined and calculated by CKJM\(^2\).

From the chosen metrics, we observed across Apache Software Foundation projects Derby, Pdfbox, Lucene and Ivy releases that file’s churn correlates with discussion, while only in very few releases a file’s structural complexity correlates to the amount of times the file is changed to address a given issue.

\(^2\)http://www.spinellis.gr/sw/ckjm/doc/metric.html
The consistent churn and discussion correlation across project releases (as it will be showed in chapter 5) highlighted that the amount of code change led to more discussion. This highlights that programming effort measured in hours does not tell the whole story. There may be other variables in the domain, such as discussion for OSS that can be missed and should be taken into account.

We should point ahead that our framework diverges from a more known and accepted framework grounded purely in quantitative methods such as the one described in figure 2. As pointed in out in [81], there is no guarantee that a correlation between an maintenance effort estimator and maintenance effort in hours can be transferred across datasets and the choice of the indirect estimator are purely anecdotal. Lastly, there have been shown motivation in previous work for seeking alternative effort estimators [13], [36], and that care should be taken since they can be domain dependent [46]. Our framework address this by building the effort estimators directly from the dataset logs by means of a qualitative method (GT) in replacement for speculation of maintenance effort estimators.

Our work is novel in multiple ways: first, to the best of our knowledge, no other work has considered the creation of a framework to obtain alternative maintenance effort estimators. This leads to an underrepresented amount of programming effort. Programming effort when it is explicitly captured is typically captured at coarse granularity (e.g. project level), and such granularities inadequate for file level analysis.

Second, our results from GT have shown that contributors may spend time in discussion over varying situations. Effort as logged by time usually only take into account time of maintaining software,
while neglecting time spent in other situations. Our quantitative analysis shown that the amount of code changed correlates and can even predict the amount of discussion related to file changes. This calls attention that time alone may be insufficient to estimate the total amount of effort maintaining a given file.

Third our obtained results are based on release level. That is, our results shown that for a given project, our maintenance effort estimators can be predicted from metrics across releases. As advantage in comparison to other works (as it will be discussed in more depth in chapter 3), where most effort estimation require a set of projects to be developed in order to be able to be used for prediction on similar projects, our framework can be used as soon as the first release is concluded by using the current release \( n \) to predict releases \( n+1 \) or other subsequent releases.

Forth, our work show that maintenance effort estimators may be available in other form in a given domain, which opens a myriad of possibilities for effort exploration, as time and money cost estimators are unfortunately still very little available in the software engineering field. This also enable companies whose process does not include logging time on a file level (such as OSS) to benefit from our results, which is not possible when programming effort time logged in hours is used.

Fifth and last, from our obtained abstraction of OSS data, we describe a much more extensive of threats to validity in conducting research with OSS data, which is mostly neglected in data mining studies.

1.3 RESEARCH METHOD

This section describes the research design employed as the basis for this work. As previously stated in this chapter, the research goal is to provide an empirical foundation upon which sound refactoring decisions may be based, by relating variation in the complexity of code to variation in effort—and hence cost.

In order to achieve this goal, we propose a framework (shown in figure 1). The framework is divided in three main phases, which are extract indirect maintenance effort data, build and validate indirect maintenance effort models, and predict indirect maintenance effort. We discuss in the following subsections which methods were employed in order to build the framework and where they are detailed in this work.

1.3.1 Extract Indirect Maintenance Effort Data

According to our literature review (chapter 3), most work have focused in using effort as measured in clock-time, and when such information was not made available, anecdotal choices have been made.
There has also been stated motivation in previous work in seeking for new effort estimators. In this phase of the framework, we hypothesis that effort estimators are domain dependent.

One immediate advantage of such hypothesis is that this phase is more flexible in choosing meaningful variables, however given that previous choices have been anecdotal, a need for a more empirical, yet flexible approach is desirable. To fit such needs, we employed GT (chapter 2).

As we discuss in GT chapter, there are many interpretations and misunderstandings in respect to the method. To ensure the validation of our results, we first explain our interpretation of the method, and afterwards apply to unstructured data logs it in order to obtain empirically, meaningful effort estimators.

1.3.2 Build and validate indirect maintenance effort models

With the effort estimators, we inspect the current state-of-art (chapter 3) to assess which models have been created. The model choice phase includes choosing the structural complexity file metrics, deciding the granularity of data, verifying the availability of datasets, collecting and managing the chosen projects dataset, and the associated threats to validity in each choice of the model parameters.

Once those choices are made based on current work, we conduct them in chapter 4. As results we obtain quantitatively one set of proxy effort estimators, and one set of structural complexity file metrics, both associated to their threats to validity.

1.3.3 Predict indirect maintenance effort

Having collected the data and assessing the associated threats, we perform exploratory, and correlation analysis to better understand the relationship of the collected variables. Lastly, we test multiple learning methods to predict empirically effort from the structural complexity metrics (chapter 5).

1.3.4 Open Source Software Case study

The previous explained phases of the framework have been applied in OSS projects on all chapters of this work, to show the feasibility of the method. We chose them due to the availability of the data, and because effort information is not available.
Part I

QUANTIFYING MAINTENANCE EFFORT PROXIES

To choose representative maintenance effort estimators, we must have a strong sense of what the data is telling us, both qualitatively and quantitatively. At the same time, knowing which methods have already been successfully applied to analyze patterns of effort in data is of equal importance. In this part, we present our results of applying a qualitative method known as Classic Grounded Theory to understand what theory lies beneath our data, and conduct a literature review of effort to understand what is the state-of-the-art practice on chosen maintenance effort estimators in software engineering. From the obtained results, we formally define our maintenance effort estimators as variables.
Well, I never heard it before, but it sounds uncommon nonsense.

— The Mock Turtle - Aline in Wonderland - Lewis Carroll

2.1 INTRODUCTION

Software Engineering researchers have traditionally been very poor at making theories explicit. Virtually all of the empirical studies conducted over the past few decades fail to relate the collected data to an underlying theory. Consequently, results are hard to interpret, and studies cannot be compared. Few scientists give thought to how theories are created. A notable exception is Grounded Theory, a technique for developing theory iteratively from qualitative data. [72, p.297, p.298].

A theory level of abstraction enables the generalization of knowledge independently of a specific time and place. A theory generalizes analytically, consequently making it possible to generalize in situations in which statistical generalization is not desired or even possible.

GT was first conceived and described by Glaser and Strauss [25] as a means of discovering theories by analyzing data. Since Glaser and Strauss’ original work, the method has advanced along the years to provide more details on how it is to be conducted in [22], [23], [24], [26], and many others. The method has been applied in many different fields. Despite being employed in software engineering empirical studies since 1999 by Seaman [70], only recently have Adolph et al. [5] discussed the implications of employing GT in a software engineering context in depth. We have also found that there is a great confusion in respect to the method interpretation and it’s usage, as discussed in the related work section (2.3).

In classic GT, the research question is: what is the main concern of the participants and how do they resolve or process that concern. For instance,
if interviews are used for data collection, they must be structured so that the interviewee, not the interviewer, leads the discussion. Interviewees are invited to discuss what is relevant to them about the topic area, on their terms [51, p. 23].

As discussed in the introduction section (1), our main hypothesis is that maintenance effort cost estimators vary according to the associated domain in which measuring effort is of interest. Given that GT proposes to identify the main concern and how it is processed, we expect it to capture pattern behaviors of which we may use as estimators. In a nutshell, GT provide us a way to navigate qualitatively in the data while being able to construct an abstract representation of the underlying phenomena. With this obtained abstract representation, we then decide what could be potential cost estimators of maintenance effort.

Different from verification methods, there is no hypothesis to be verified in GT; instead hypotheses are suggested based on the patterns of observed behavior. GT is a way for a researcher to take a panoramic view of all the behaviors within a community, and to identify which is the most important one, based on its occurrences in different contexts. We found that the characteristics of OSS make it an easy fit for the GT methodology, since, unlike most industry processes, OSS development largely occurs in a distributed online environment, and therefore most of what happens is logged. This diminish the threat of being biased towards the wrong main concern, which could occur in interactions that are not logged.

We employed the GT methodology on the Apache Lucene¹ project, and later expanded our investigation to include Apache Jakarta². In the results section 2.5, we discuss which projects were chosen as was required by theoretical sampling. We opt to present them this way, to enforce that there is no pre-selection of the data that will be used. Theoretical sampling will be explained in section 2.2.

The main contributions of this work are as follows: First, to our knowledge, no previous work has applied GT in OSS communities. Therefore, our results provides insights on how theories in GT are shaped for the OSS domain. Second, most GT methods have only applied GT rigorously by conducting interviews. We explain the challenges in applying GT to logged data. Third, we have found that GT provides great insight on data quality of open source projects, which we also present in this work, extending the notion of threats to validity in conducting empirical studies in OSS datasets. Forth and last, our interpretation of GT differs in some points as discussed by Adolph et al.[5]. Therefore, we contribute by extending some definitions from GT, and discuss more in depth how our interpretation differs from existing works.

¹ http://Lucene.apache.org/core/  
² http://jakarta.apache.org
In section 2.5, we described the results in conducting the method. In the following section we explain the major stages and building blocks of the method.

2.2 CLASSICAL GROUNDED THEORY

In this section we explain the main terminology underpinning classic GT. We illustrate the terminology with software engineering examples drawn from our results. This section follows from our understanding only from works published by Glaser in GT and our experience gained in applying it. Although we initially found the terminology of GT confusing, we do believe its complete understanding is necessary to generate a theory. Furthermore, the differences among the terminology used in GT is one of the hardest concepts to understand from the method. We described it here as a set of building blocks, how they are transformed, and under which phases they appear. Such distinction is not clear in GT texts.

Our understanding is that a researcher interested in conducting the method should be able to build the big picture of the method after reading back and forth all the texts, participating in seminars or being mentored by an expert. Perhaps, this is one of the biggest drawbacks in minor mentoring, that is, self teaching the method. Due to this, we consider this definition subsection a contribution on its own, in the same fashion as Adolph et al. [5] which devotes an entire paper. Our interpretation differs in some points from Adolph et al., as we will highlight in the related work in section 2.3, which is why we felt the need to explain our interpretation of GT. Finally, since we noticed there are varying definitions of GT over many different fields [24], we thought that instead of judging our interpretation correct, we would state it in detail, so that it can be judged as valid or not based on other interpretations of the method.

2.2.1 Grounded Theory Building Blocks

In this section we highlight the building blocks of GT. The building blocks are highlighted in italic during this whole subsection. We explain in detail the constant comparative method and theoretical sampling stages in the next subsection. We opted to explain the building blocks mostly by examples because without context they feel ambiguous. Understanding these building blocks is required to understand GT phases discussed in the upcoming session 2.2.2. Together, they show our interpretation of the method, which can then be used for assessing the validity of our results. This section definitions also constitute a contribution in showing how each building block and phase of GT can be mapped to log data, specifically OSS data. Some methods will be explained in this section as black boxes, such as the constant
comparative method, highlighting which building blocks are input and output of them. In the upcoming section 2.2.2 we explain their details in order to avoid confusion.

**Incidents**

An *incident* is an issue in JIRA, a file log of IRC chat, or a thread of e-mail in our data. Broadly speaking, it is a unit of document that describes the phenomena on its raw form.

**Line-by-line analysis**

*Incidents* are analyzed *line-by-line*. In our understanding, a "line" is defined by the minimal amount of sense it makes to the researcher. By analyzing "line-by-line" we ascertain what the authors are saying without inputing what was said, interpreting it, or reifying its meaning [23, p.24]. The "line" thus may be an entire paragraph, or just few words. Although analysis without interpretation may sound strange, the point made by the author of the method is that in reading line by line, the researcher is not able to fully understand the whole situation. Instead, only fragments of the text are read, which makes it harder to make sense of the pieces. Other parts of the method aid in storing, organizing and analyzing each *incident*.

**Indicators**

Among the "lines", some are more important than others. Those that are more representative are called *indicators*. These *indicators* may vary significantly. For instance, when the *incident* is an JIRA issue, it could be a error dump, or someone requesting opinion from other people involved on solving the issue, and so on.

We can label the *indicators* by *coding* them. *Coding* can be understood as a renaming of the *indicator* that still captures its essence. For example, an error dump spam over many lines and had little meaningful information to be analyzed since it required knowledge of the code and the problem itself which goes beyond the *incident*. Such error dumps, although considered *indicators*, were more meaningful by being *coded* simple as "error-dumps". Since part of the constant comparative method requires to constantly compare *indicators* and *codes* among themselves, it would be much easier to compare the short expression "error-dump" rather than the whole dump itself against other *codes* and *indicators*.

**Codes and Coding**

The generated *Codes* can be either *substantive codes* or *theoretical codes*. Both codes are generated by the constant comparative method. Since a *code* is the result of the constant comparative method, then the *code* represents an *indicator* of an *incident*, and as such is representative of an aspect of the underlying phenomena in the data. This is a conceptualization of the underlying phenomena, and is a shared property between both the *theoretical* and *substantive* code. What makes them distinct is the kind of conceptualization they represent, or in other words, how they abstract the phenomena. A *substantive code* is more closely related to the data. When we mentioned the "error-dump" *code*, such code was in fact a *substantive code*. A *theoretical code* conceptualizes how the *substantive codes* relate to one another as a hypothesis to be integrated in the theory. By means of the constant com-
parative method, substantive codes saturate into categories and properties, both which can have dimensions. Since categories and properties are just saturated substantive codes, theoretical codes also relate them. An example of a theoretical code is: "Solution Aids are provided and discussed iteratively in seek for a consent over a solution to piped problems". This theoretical code relates the category piping problem and the property Solution Aids of Iterative Informed Consent (IIC) by explaining that they are used as aids.

It must be made clear how these hypotheses are generated. For example, one theoretical code that was used for IIC was of informed consent (described in detail in section 2.5 - Results). We initially borrowed our understanding from the pattern behavior of "informed consent" between a patient and a doctor in medical sciences, and we adapted this theoretical code so that it fit our understanding, grounded in the data of the pattern behavior observed in OSS, by adding the term "iterative". That is, theoretical codes can be seen as a way to encode phenomena in words. These codes can then be reused in other substantive areas. By increasing his knowledge on theoretical codes, the GT practitioner can better identify and use a pattern borrowed from other fields. For instance, during our analysis we borrowed a pattern from medical sciences known as informed consent. Informed consent was then adapted to our theory, which resulted in IIC. In that sense, theoretical codes can be borrowed from observed patterns in other fields and considered under other contexts that at first couldn't have been thought of (such as OSS!). In Glaser [22], Glaser describes just a few of such codes. It is up to the researcher to search, learn and reason about using more theoretical codes so that the relationships within the data make sense. While this enriches the method, it makes it more subjective and perhaps subject to misunderstandings. The use of theoretical codes is also what motivates literature review in a problem *not* related to the domain where the grounded theory is being applied to be performed. This way, the researcher is not biased into attempting to verify existing theories, and still is learning new theoretical codes that can be applied. We discuss the role of literature reviews in GT in the next subsection (2.2.2). In short, theoretical codes encode in few words a kind of behavior pattern in the data that is representative of phenomena that can be borrowed from other substantive fields or created during the analysis.

Theoretical saturation of substantive codes occurs based on the data collected and analyzed by the researcher. As the researcher continuously performs data collection and analysis, more of the phenomena is coded. The substantive codes evolve by the comparison of indicators and codes resulting in concepts. There is no precise moment of when a substantive code saturates into a concept. The "gut-feeling" is that the researcher has already observed a significant amount of the phenomena is captured in a given code. At such point the code label (that
is, it’s describing sentence) has already captured the phenomenon for the researcher. For example, at a later point in the analysis, it suffices to talk about IIC instead of continually applying the constant comparative method which will output IIC repeatedly. From a software engineer perspective, this can be compared with how developers discuss design patterns. Instead of redefining its characteristics, the name of the design pattern is used as a common vocabulary to express the intent of a design. In GT, the labels are used in the same form, but for explaining the underlying behavior of the phenomenon. Finally concept saturation leads to the creation of its properties. Properties explain variations of a given concept more explicitly. For example, we know about what IIC is, which is in short, the continuous iterative discussion process seeking for consent. However, we also know that during IIC there is a constant offering of different ways to suggest aids to reach the consent. This additional variation of the phenomena is encoded in Solution Aids, a property of IIC.

Concepts can further saturate into categories. Our understanding of the difference between concepts and categories is that the saturation of a concept will lead to the creation of more properties, and its observed dimensions. A dimension can be understood as the range of the observed phenomena. For instance, the kinds of Solution Aids that contributors in OSS provides during discussion of a problem. Thus we can think about dimensions of a property. We can also think about the different kinds of IIC in the analyzed phenomena. Thus, we can think of dimensions of a category. Furthermore, a category may also include many concepts. We observed this mostly during theory reduction, that is, when we reduced many concepts into one or few categories that explain the same behavior. Theory reduction can be seen as a parsimonious way to capture the phenomena using a different set of categories that is reduced in comparison to the initial set of categories. One can imagine this as one person who explains in a very complicated way, a situation, while another person describes the same situation in a much more concise way. Both people describe the same situation, yet one is more parsimonious than the other. In this way, a category may end up including many concepts that were reduced. One does not necessarily need to reduce concepts to observe a category as being a group of concepts. Glaser [22] describes other situations in which a category may contain a set of concepts, but in our analysis we mostly observed this during reduction.

Among the categories one or more will be identified to be the core category. A core category is a behavior pattern that is manifested in different ways and relates to most other categories. For example, IIC is intuitively something that should be expected to occur in OSS and manifest over many different situations, as most open source is based on collaboration, and most of the time it is voluntary. For the report of the results of the GT, however, only one category is chosen as the core
category. Each core category consists in a set of results on its own right, as each of them would demand explanation of itself and the categories that most relate to it. It is not a strict rule that in a given analysis only a single core category exist, however during the analysis one will be chosen as focus as more data is collected. This can be imagined as having to choose a route during the analysis for continually applying theoretical sampling. In choosing a route, one core category will prevail against the others. The researcher can for another study “go back” and choose another route, further reporting the results of other core categories.

Memos manifest the emerging understanding of the data, as it is being formed by the researcher. Memos will also be a source for theoretical sampling, identifying concepts, properties, categories and dimensions. They work as a backup memory for the researcher, and will serve as reminder of ideas regarding a code, concept or category, and helping on their saturation.

### 2.2.2 Grounded Theory Phases

In this section we describe the major steps in conducting GT using the previous defined terminology. We highlight how each stage affects and relates to incidents, indicators, concepts, categories, properties, dimensions, etc. Sometimes we will restate some of the definitions briefly, but now giving emphasis to the stages in which they are generated. Furthermore, for the remainder of subsection 2.2.2 we will display in italics the main steps in conducting GT instead of the building blocks as in subsection 2.2.1.

In Classic GT, the practitioner may shift emphasis among various stages ranging from theoretical sampling or data collection to analyzing and memoing. When there are lots of memos the focus may shift to sorting. Sorting may indicate a thin area, while writing and rewriting may require more constant comparisons. It must be stressed that the ordering of these terms carries no weight: this is not a linear process. Instead the stages can be see as a cycle in circles, where few stages are emphasized while others fade away [23, p. 15, 16].

The entire process is guided by the constant comparative method and theoretical sampling. The constant comparative method expresses the need for the practitioner to constantly compare each generated code against every other one. It is by this comparison that memos are also generated, which may in turn be compared as well. Such comparison is what leads the GT practitioner to understand where patterns overlap, relate, or are distinct. It is also by means of the constant comparative method that codes saturate into concepts, and later into categories. During the analyses of the data, the researcher should constantly ask himself: "What is this data a study of?", "What category does this incident indicate?", "What is actually happening in the data?", "What is
the main concern faced by the participants?”, “What accounts for the continuing resolution of this concern?” [22, p. 57].

Analysis of the data consists of many parts. Substantive coding occurs by labeling paragraphs. Such labels should shorten, while still expressing the meaning of, such a paragraph. Substantive coding is divided in Open and Selective coding. **Open coding** occurs during the initial phases of GT where a core category has not yet emerged. **Selective coding** occurs after the appearance of a core category and is focused on supporting this category. This approach does not bias the analysis: at the point of the selective coding, theoretical saturation (that is, when the next piece of information points does not reveal a new aspect to be coded, and adds nothing to the theory [25, p.111]) should have already occurred, so instead of continually coding the same thing over and over, the practitioner should focus on what behavior pattern seems to explain and relate it to other pattern behaviors.

In our experience the process of theoretical coding occurred mostly implicitly in our memos during the constant comparative method. When memos were assessed later they helped integrate the theory by generation of theoretical codes. As Glaser [22, p. 72] points out, one talks substantively and thinks theoretically of the relationship between codes.

The GT practitioner is constantly asking what to code next. This is guided by theoretical sampling. The idea is that given, for example, little evidence of a category, a practitioner may seek further places where that information is available. Alternatively, divergent information related to a category may also be incorporated to evolve the theory. Such incorporation of divergent information that must be accommodated by the evolving hypotheses. This is one of the many aspects that distinguishes GT from verification approaches. There is, thus, a constant process of adaptation and evolution of the abstraction of the observed underlying phenomena, grounded in data. An extensive example of how GT evolves using theoretical sampling is given in [25, p. 59]. **Theoretical sampling** is a different way to sample data for acquiring further evidence than the most conventional random sampling in statistics. Instead of randomly obtaining new incidents, the GT practitioner decide on which incident we will search next based on the saturation or memos created during a given moment of applying the method. Since theoretical sampling requires understanding what concepts are still thin in the analysis, it requires the constant comparative method to generate concepts so the researcher can assess where to go next to obtain data. On the other hand, the constant comparative method must continually have more data to be compared. So one method complements the other.

**Theoretical Sampling** also varies according to the stages of GT. More specifically, in early stages we seek to saturate the categories, and in
later stages the *theoretical sampling* is more targeted in the core category. During interviews, this process may seem more intuitive as once the researcher hears from the interviewee he is able to interact more towards their problems in the environment. When the GT practitioner data is available only as logs, the GT practitioner can’t just ask the dataset about its problems.

*Memoing* consists of instantaneous ideas that occur to the researcher while coding and understanding the data. The process of memoing is seen as a way to not lose any interesting idea, by constantly taking notes as soon as the idea occurs, for further reference and comparison.

Memo sorting is an exercise of sorting thoughts about the current state of the analysis. In early phases this can be useful to stop and think over what are the currently captured phenomena, and non saturated concepts that may be in need for theoretical sampling. In later phases, *memo sorting* is as a way for the practitioner to organize memos towards the practitioner final writing; the end result is typically an outline for the final writing of the results. In the appendix A we provided all our memos after *memo sorting*.

A Literature Review in GT should only be conducted in late phases of the work. This constraint may seem confusing at first, but it is related to the generation of theoretical codes. The researcher should not conduct a literature review on an early phase, as this may cause bias. It is, however, considered acceptable to do a literature review in the *memo sorting* stage as, by this point, the categories have already been created and the literature review may help in generating additional concepts to code [23, p. 67].

Finally, it is also important to note the conceptual level of the analysis. It is by means of the constant comparative method that our codes, and thus our understanding, evolve from merely being labels over the data to become more abstract concepts and categories. In terms of our study, this can be understood as raising the level of abstraction of the pattern behaviors observed within the OSS community and how they relate. As we raise the conceptual level, together with *theoretical sampling*, this may lead us to obtain new data to analyze a set of evolving concepts. Conversely, we may merge concepts if we are interested in maturing the existing categories. Thus, when beginning the generation of a substantive theory, the researcher establishes the basic categories and their properties by minimizing differences in comparative groups [25, p 56]. This process of abstraction in turn generates labels, indicators, categories, properties, dimensions and so on. As the abstraction level rises, different names are given to the pattern behaviors and their structure, (codes, concepts, categories, properties, dimensions). It is also by continuously raising the conceptual level that substantive theories can evolve to formal theories (that is, that explain pattern behavior across substantive fields).
2.2.3 Conclusions About the Method Interpretation

During the whole process, when we were conducting GT, the previous defined interpretation is what kept us from drifting away from the rigorous of the method. The line-by-line characteristic of the method constantly reminded us to not try to interpret the data. The difference between substantive codes and categories reminded us that there might be underlying phenomena not yet captured in our analysis. Theoretical Sampling reminded us of not trying to go after an interesting dataset related by literature. The constant warning of not doing a literature work before entering the late phases of the method was mostly rewarding when we found that some existing theories such as the benevolent dictator did just not appear in our analysis. Had we known about such theory beforehand, we might have be tempted to sample after seeking such theory, believing that it was missing in one of our categories, thus not making it saturated enough.

In further sections we also explain how we slighted adapted the method to our needs. For instance, GT is suggested to be conducted using paper, pen annotations and scissors, which could be substituted by other tools in a computer. Still, such small divergences from the original method should be made clear to assess the validity and contribution of this study. For an in-depth understanding of GT, we suggest to start with Adolph et al. for a pragmatic discussion of the method as it applies to software engineering, followed by the writings of Glaser, specifically: [25], [22] and [23] and their referred works.

2.3 Other Grounded Theory Work

We have found little work in software engineering that has used or adapted Classic GT. Furthermore, we noticed that our understanding of the method diverges in some parts from related works. Since these work do not conduct GT in the same domain as ours, that is, OSS communities, we devote this section to highlight divergence of the method understanding in software engineering and against ours. We do so as to show how the work of Glaser and Strauss can be conducted in online environments with logged data. In particular, previous considered disadvantages of the method may be turned into advantages. This is due mostly to the limited available technology of when the method was first published.

Adolph et al. published four works related to GT being applied to software engineering, where two of them were extensions of previous work. In [4],[5], the method explanation, challenges and rewards of applying GT to understand software engineering problems are discussed. In [3], [6], the focus is on explaining the results, with the method only explained briefly. GT is applied to investigate how peo-

ple reconcile their differing perspectives and manage perspective mismatches in an Agile software development project.

Our interpretation of the method varies from Adolph et al. in that they recommend using both semi-structured interviews and direct participant observation because, according to them, interview data alone may provide a superficial description of a phenomenon and accesses only what participants say they do. They also mention that without participant observation, it is likely that they would only generate a theory of how people create software and not how they manage it. These guidelines would render approaching data in online communities superficial without conducting interviews (e.g. by email). We argue against this by referring to Glaser’s and Strauss’ first work [25, p. 161], where a whole chapter is devoted to “New Sources of Qualitative Data”. In this chapter, Strauss says that the practitioner is encouraged to use both kinds of data based on their advantages and disadvantages. A quick scan over the disadvantages made it clear that they do not affect the type of data we have. The advantages of log data (cited as “library data” in Glaser [25]) are of accessibility (the narrative may be from someone who already passed away or being able to conduct the person again in later phases of the work or the practitioner own agenda, leaving gaps), spatial obstacles to reach the data are less than for interview, willingness for the participant to be observed or interviewed. All together those advantages may be seen as of effort, cost and speed for data gathering. [25, p.177 and p.178]. The disadvantages of approaching data in a library are that (1) Certain groups of people may simple exist and vanish without leaving registered documents, (2) Dense detail of the data such as issues, relationships, roles strategies and processes, (3) Data can be purposely misleading, and lastly (4) written reports of available events may be deficient of the real event occurrence [25, p.180 and p.181] What is important to note is that our data is not authored by someone such as in a library, but, instead it is logged. As such, we extend Adolph et al. argument on previous works that the type of data may require or not the need of interviews for generating substantive theory.

Hoda et al. [30] applied GT to better understand self-organization by presenting agile teams. They describe their results mostly by example and structure of obtained categories. They used face-to-face semi-structured interviews with open-ended questions. Questions were targeted experience working in Agile methods and in their roles on Agile projects. We found no mention of theoretical codes in their interpretation, theoretical sampling, used tools or core category. This might explain why the results are more focused on the categories description and explanation, rather than categories relations by theoretical codes.

Power and Moynihan [65] used semi-structured interviews to obtain a theory of requirements documentation situated in practice. An-
alyzes were conducted using ATLAS.ti\(^4\), a qualitative software research tool. The results are focused on coding, classification and descriptions rather than the generation of a theory in despite of the work title. For instance, we were unable to find mention of a core category or theoretical codes integrating the analysis.

Treude and Storey \([74]\) uses a different \textit{GT} approach from us \([18]\) focusing in knowledge management, in particular, how they can be effectively deployed in software project. This method is mostly easily identifiable by the \textit{axial coding} phase which does not exist in Classic \textit{GT}. Quantitative data acquired by web scrapping was used for initial understanding, followed by semi-structured reviews.

Whitworth and Biddle \([79]\) also applied \textit{GT} in agile teams. They were interested in exploring the socio-psychological experience in agile teams, where agile teams were viewed as complex adaptive socio-technical systems. Interviews were used for data collection. Despite references to Glaser’s works in Classic \textit{GT}, the authors explicitly define in their interpretation the \textit{axial coding} phase, which suggests they were possibly using Corbin and Strauss’s \([18]\) work.

Schoonewille et al. \([68]\) uses an adaptation of \textit{GT} only during the analysis to better understand the use of software documentation. Since think-aloud protocols, experimental settings and other non-related \textit{GT} steps are used, we disregard any further comparison of this work.

Lastly, we found a particular interesting adaptation of \textit{GT} \([69]\) which was used to obtain a taxonomy of affect in collaborative online chat. We noticed again that the used method was the same as used in Corbin \([18]\). This divergence is made even more clear in that refinement of an initially hypothesized theory is accepted, which is not the goal of Classic \textit{GT} as explained in the introduction (2.1).

Overall, we found that all works that did not deviate from the method itself focused on using interviews as their main acquisition of data as an input to the constant comparative method of \textit{GT}. This further enforces the contribution of this work in highlighting the possibility of attaining meaningful results in applying the method as fully described solely on “library” data in modern systems, more specifically on log data. Furthermore, we observe that most works do not fully use the method, but instead either focus more on open coding or on adapting the method. This may cast doubt on whether or not a theory is in fact being generated, as the method procedures are not fully followed. This is another motivation to clearly state our interpretation of the method and why we believe we were able to generate a substantive theory of \textit{IIC}. Lastly, we claim that little work in \textit{GT} has been applied in software engineering. Even among the little work we have found, only Adolph et al. explicit cite that \textit{GT} has been applied in their title, which makes it difficult to identify such research.
Since our analysis was conducted on an Apache OSS project, we first considered the set of artifacts that could potentially be studied: JIRA issues, chat logs, commit labels, and mailing lists. We decided to start with Lucene for convenience, since their chat log is publicly accessible.

In conducting Classic GT, we should be guided by theoretical sampling, as previously explained in section 2.2. Despite the selection of Lucene for convenience, the remaining projects in Jakarta were chosen based on the observed emergent categories. To manage the scope of this work, we decided not to move beyond the Jakarta project—with its rich set of sub-projects, tools, libraries, and frameworks—to avoid raising excessively the conceptual level, thus leading to the creation of many thin categories.

We observed that the categories started to stabilize when we had analyzed around 100 Lucene issues. We then moved to the other sources of data in Lucene. Approximately 2 months of each source of data from Lucene were analyzed. Furthermore, the sub-projects Slide, Alexandria, and Cactus were sparsely scanned as, at this point, the IIC category had already emerged and, although we had codes indicating that project survival was dependent on the community, no clear evidence of survival concern were indicated. Upon moving to such projects that were retired due to a variety of reasons, we were able to better understand how a concern for survival related to IIC. Upon saturation, no more data was sought. Thus, the final dataset was defined as the set of sub-projects of the Jakarta project.

We addressed the issue of "what to sample next" in a scenario without any interview data by starting from JIRA issues, and moving initially over time as issues were posted. After the first concepts emerged, we then started to vary project and communication means. In particular, as will be more in-depth explained in results (section 2.5), we decided to move over different communication channels when we noticed that a problem would be "follow-up" in other places and how these "follow-ups" influenced the problem solving. We also changed projects when we noticed indices of survival discussion but which were sparse in Lucene.

Although we closely followed the classic GT methodology, we did have a (minor) point of divergence. The method, as originally conceived, suggests using using paper, scissors, and other materials for collecting and organizing data. Consistent with Adolph et al. [5], we did not follow such approach, but instead used Microsoft Word, as shown in figure 4. Each comment represents an annotation. We manually distinguished codes from memos by adding a prefix of either "C:" or "M:" respectively. "M*" memos were annotated to represent

http://colabti.org/irclogger/irclogger_logs/Lucene-dev
Figure 3: Iteration Blocker Branch. Lack of time is one of the properties of iteration blocker. We dispose the codes over a mind map to organize the codes. The core theory (not shown) remains as the root on its final version, leading to different branches (related categories), one of which is Iteration Blocker, having as one of its properties the Lack of Time. Text information that generated the property are exposed as leaves.

Figure 4: A coded JIRA issue report. In a word document, comments were used for memoing and coding. Memoing and coding on each document was distinguished by using the prefix ‘M:’ and ‘C:’. This made it easier to query the memos after they were moved to mind maps, for organizing them to the memo sorting step.

memos of memos. Such memos were created during comparison and thus do not have an incident mapping. Figure 3 displays one of the branches. A list of the memos and the memos of memos after sorting is provided in appendix A.
As the means of communication is freely available online, we convention on copying and exporting each issue, file of chat log, commit label as a file in a local computer folder. Our dataset is thus comprised of any available textual information on the cited means of communication. Not all analyzed files were downloaded, only those who were annotated. This results in the collected files being representative, and yet not consisting entirely of all the analyzed data. The reason for this decision is that incidents that were not adding any information to patterns already found, other than supporting it with more evidence (in a frequentist sense) were unnecessary for the method.

Again, GT patterns are based on comparison and not on the relative frequency of occurrence. Instead of collecting evidence for supporting results by frequency of occurrence, which is not part of the method, we decided to browse across the incidents in seek for more evidence with theoretical sampling after saturation. Since we only did not archive evidence that were seen as completely redundant, one should be able to find mapping from any category or concept and it’s associated properties to incidents and indicators.

For the memo sorting and for the constant comparative method, we found it useful to use a mind mapping tool, where nodes were structured as categories and their branches contributed to its structure, finally resulting in leaves which represented the fragment of text to which the label refers. This differs from Adolph et al. [5] as they used a Wiki. Figure 3 shows one such branch for the "Lack of time" property of the "Iteration blocker" category. As can be observed in the figure, our convention was that the file name was the same as the label so we could map back to the source during the analysis.

This mapping from theory to incidents is provided by five versions of mind maps that capture the concepts and its saturation to categories. Each category node have many branches, each branch annotated with a code and a file name of an incident comment. Thus, mapping incidents, indicators, codes, concepts and categories over 5 points in time. Figures 3 and 4 display a branch on one of the mind maps and an JIRA issue file after downloaded respectively. In those two figures, one can see how the mapping from a property of a category can be connected to an incident by their labels. Finally, a category or concept is connected to a property directly on the mind map figure, which is only not shown here due to space constraints. The full tree over five evolving snapshots, and the incidents with the log of all modifications performed on each file are available in the online report for the associated chapter.

Since all the analysis is still all being done manually, it is our belief that using such tools does not compromise the analyzes, as we do not attempt to automate the analysis itself, but only change the supporting infrastructure for it to be performed.

6 https://github.com/carlosandrade/undergraduate_final_project
2.5 RESULTS

In this section we discuss the generated substantive theory of iterative informed consent (iic). In the discussion that follows we will adopt the convention of underlining properties and using spaced small low caps for concepts and categories. Specific details of properties and theoretical codes will sometimes be emphasized in italic. In the following subsections, we will introduce each concept and its properties, followed by how they relate to one another. We conclude by discussing iic as a Basic Social Process. Figure 5 uses the model theoretical code [22, p. 81] to illustrate the categories and associated relationships that will be presented in this section.

2.5.1 Chance Need

chance need is a stochastic pattern behavior that generates piping problems to be solved by iic. It is not known when new piping problems will occur. They originate from different piping problems source entities, whose motivation exist, yet are rarely stated. Due to this, piping problems are usually assessed by the community based on their reproducibility. If they are able to be replicated by other members of the community, then they can be processed by iic. Furthermore, source entities vary vastly and can potentially be any entity interested in the community. Source Entities can be represented by a single person interest or a group of them. A piping problem...
generated by \textsc{chance need} does not necessarily flows within the "pipe of concerns" of a community. \textsc{iic Concern Acknowledgement} must occur in order for it to be solved. We found that a theoretical code from microeconomics, \textit{the principle of the invisible hand}, explains the connection of this variation over the Chance of Need from both parties, as a stimuli for evolution: \textit{It is by individual seek of maximizing gain that the society maximize welfare.} Where welfare here is the problem solving by \textsc{iic}, which results in gains for both sides. If one side does not observe gain, then the principle is not being applied, as the problem is never acknowledged in the first place.

2.5.2 \textit{Piping Problems}

\textsc{Piping Problems} are named after the pipe-and-filter architecture pattern in software engineering. \textsc{Piping Problems} after being generated by \textsc{chance need} may or may not be "piped", depending on their \textsc{Concern Acknowledgement}. If they are "piped", then they will undergo many filters, all of which are different forms of \textsc{iterative informed consent}. Both \textsc{piping problems} and \textsc{iterative informed consent} evolve over time, until the \textsc{piping problem} is terminated.

\textsc{Piping Problems} may coexist in different \textsc{communication channels}. \textsc{Communication Channels} impose constraints over how \textsc{iic} occurs to process \textsc{piping problems} by having different kind of participants involved and the type of \textsc{iic Solution Aids} they provide. All \textsc{communication channels} usually converge in a single one where any kind of \textsc{Solution Aids} can be provided. Piped problems may be processed over different \textsc{communication channels} depending on \textsc{iteration blockers} follow-ups or \textsc{piping problems} source entities. Its termination may therefore occur while co-existing in different \textsc{communication channels} or converging into a single one.

\textsc{Piping Problems} can either be of logical or \textit{technical} type. Logical problems are those where the functionality is misunderstood. This misunderstanding occurs due to the external participant background or wrong documentation. \textit{Technical} problems are those which there was a mismatch between what was intended to be implemented and what actually is implemented.

\textsc{Piping Problems} \textsc{Concern acknowledgement} may lead also to \textsc{iteration blockers}, which may occur due to required effort, lack of time, refusal of responsibility and follow-up discussions. \textsc{Piping Problems} \textsc{Source Entities} also impact over \textsc{iteration blockers} by their \textit{activeness}, impossibility of replication, expectation, and understanding conformance.

\footnote{http://en.wikipedia.org/wiki/The_Theory_of_Moral_Sentiments}
2.5.3 Iterative Informed Consent (IIC)

IIC mainly occurs as a way to solve piping problems, assess if they still exist, assess the health of the community, releases, and as a good (service). Each such manifestation of IIC can be seen as a filter that process the piping problem under varying communication channels.

The generation of problems by chance need are processed by IIC in OSS employing continuous discussion until a solution is reached. IIC motivates a consent-based solution by the majority; an improvement over previous solutions based on a continuous process of stating and analyzing facts. Solutions are added and discarded and this is understood by the community as part of the search for optimal choices. This continuous discussion to solve a problem is pattern employed by the community over many different situations and it is even at times perceived as a good (service). Continuous discussion seeks to clarify risks and potential solution benefits to address the piping problem. This discussion continues until potential solutions emerge and one is chosen, either informally or formally based on voting by the participants. One theoretical code that we observed from medical sciences seemed to fit well to such a behavior, which is informed consent. According to the Oxford Dictionary, informed consent is defined as

"permission granted in full knowledge of the possible consequences, typically that which is given by a patient to a doctor for treatment with knowledge of the possible risks and benefits."

For this theoretical code to earn its way in our theory, we noted that, different from informed consent in medical sciences where the doctor fully understands the risks and benefits and the patient is attempting to understand as part of the consent process, in an OSS community such a process does not flow downwards (from doctor to patient) and no-one holds full knowledge of all risks and solutions. It is by iteration that risks and solutions arise and saturate for piping problem solving. Thus, IIC represents this continual generation of risk comprehension, and solution creation until saturation is reached among participants (although not necessarily total agreement) and a solution is elected, either formally by voting or informally by a decision of the committer.

In IIC, solution agreement and disagreement continue until saturation. In parallel, varying degrees of motivation manifested by sourceentities affect the participants’ response times and participation. IIC also relies on the participants being willing and eager to learn from others’ opinions:

Lucene-DEV-19-C: Anyway, I'm very curious as to how your directory caching code works.
Lucene-DEV-48-C: I am really interested in feedback. First, do the APIs work for your needs? Also, does everything work? What kind of performance you are seeing? Are there things that could be done better (especially in terms of file structures and reading of those files, I think this is where the next layer of optimizations should come from).

In addition, IIC relies on participants being open to feedback:

Lucene-DEV-16-C: Could you give me some pointers to the approach you think should be taken on this one?

As solutions mature towards saturation, **Solution Aids** are provided by **Piping Problems** source entities and the community. They may take the form of unit tests, code snippets for bug replication, experimentation and feedback (reports from users who have already attempted some usage and its results), crash examples, error outputs, etc. They are also influenced by **Piping Problems communication channels** by limiting the types of aids that are possible to be made.

**Solution Aids** vary in time as the understanding of the **Piping Problem** that is being processed by **IIC** improves. For example, when little is known about the problem **IIC** usually revolves around experimentation and feedback, crash examples, error outputs and unit tests. Later, as the problem becomes better understood, **IIC** generates solutions examples and solutions attempts, usually reflected as patches.

**IIC** also manifests itself as Partnership, Bargaining, or **Concluded**. Partnership **IIC** are unique in that the **Piping Problem** are thin and usually involve formal decisions. **Concluded IIC** occurs when Pipe Problems are reported by source entities with a plausible solution that does not require subsequent saturation and consent. Bargaining occurs when an external party directly benefits from the **IIC** itself, thus seeing it as a good, and attempts to convince the community:

Lucene-DEV-41-C: I have to apologize for so many messages to the list, but I really have to get the TermVector stuff working within the next few days because the next release of our application is going to depend on it. Once the release is done, it will be much harder (for us) to make changes to file formats. So I’m going to continue being a bit noisy for a while, just so everyone has an opportunity to comment on the changes as I’m making them, and so we don’t have to make too many changes later on. With enough input, the result will be a whole new set of capabilities for Lucene that everyone can use! Isn’t that cool?

---

8 emphasis given by the authors.
IIC bargaining and its defined properties can be perceived as a marketplace. The underlying pattern is that a game of convincing of welfare for both sides, as previous referred on the theoretical code of the invisible hand, occurs in such market, where the good (service) is IIC.

2.5.4 The IIC Marketplace

As previously discussed in IIC (2.5.3), one of its manifestations occurs as a service, which has been observed to be a manifestation of the invisible hand theoretical code.

The concept of the marketplace is driven by interest. If interest does not occur from both external and internal parties of the community, a piping problem is never generated. Interest may also be manifested as personal or collective, over interest inquiries. Depending on the degree of interest from the involved parties, contributions may vary from suggestions to direct contributions. The bargaining surrounding the contributions is usually affected by the expectations of effort and time tolerance for solution. If the problem is urgent, then the external party is typically responsible for a concrete contribution. If, however, the effort is small and community interest is high, external parties may take a more passive role. In summary, as the perceptions of effort and urgency vary, on the part of all parties, some will take a passive role and others will take an active role. In both cases IIC is a service, since feedback is requested to get the problem solved.

2.5.5 Iteration Blockers

During IIC, there are varying types of blockers to the iteration. That is, even if a piping problem is not fully processed by IIC, different factors may block the continuation of the IIC. Those blocking factors are influenced by piping problems source entities and as a response of the IIC concern acknowledgement judgement by the community.

Piping problems source entities may provide impossible to replicate problems which, as mentioned before, will not be piped as they are not reproducible. This blocks IIC until further details are provided by source entities. Furthermore, the source entity activeness play a major role in blocking the problem processing. Long time periods without providing feedback may lead the piping problems to end up in an unknown status which will require IIC to be used to assess if has been solved or not.

Lucene-4-C: This bug is over a year old, and we have not heard from the original bug reporter, so I'm closing it.
There may also be an misunderstood expectation by the source entity in respect to what the community is interested in addressing. Thus, wrong expectations will lead to a refuse of concern acknowledgement, further blocking the problem processing. Understanding of Conformance blocks the iteration towards a solution as there is no problem to begin with in such cases. This is a representation of logic piping problems (2.5.1). In this instance, source entities believed that there was a problem, when in fact it was misunderstood what a certain functionality was supposed to provide.

IIC concern acknowledgement may also block the process of the problem. A problem may be acknowledged, however not processed due to estimated effort to process it by the community. In such instances, source entities are expected to provide significant contribution towards the problem processing to trigger its solution. Lack of time may also keep community to process problem. Refusal of responsibility is directly influenced by concern acknowledgement, being one plausible explanation of what the community judges a concern that can justify IIC. Refusal of responsibility is a strong manifestation of the invisible hand theoretical code for IIC. Finally, follow-ups that change piping problems communication channel may also manifest as iteration blockers. They influence the motivation of piping problems source entities of further requesting the problem by chance need since they must be restated and have already received a negative concern acknowledgement as in Lucene-40-C:

Lucene-40-C: Since we established that this is not a bug I will close the bug here. If you need more help, please use Lucene-user list.

Denial of report is an where the means of communication affects the involved parties’ behavior. Depending on where the chance need is manifested, the community might defer the request as it has been posted in an inappropriate place. Denial of Report is subject to time, as it has been observed that the justification for denying reports begins to be accepted due to a change of internal rules within the community.

Lastly, a piping problem may remain unsolved due to false judgment of being solved. This further reflect on its manifestation over many releases and issues.

2.5.6 Uncertainty of Conclusion

Resulting from iteration blockers activeness, these are frequently problems that are seldom solved and no action is taken until many months later. The long time period occurs in the hope that whatever blocked the iteration disappears and the process can continue until
solution saturation. This long period of time sometimes causes uncertainty over the status of the problem. This uncertainty is addressed by IIC iteration over uncertainty in requesting opinions and facts on finding whether or not the problem has been solved. Uncertainty of Conclusion may also be solved by self verification:

Lucene-116-C: Judging from sizes of .java files in this attached ZIP files and sizes of files in CVS, it looks like this is already in CVS.

Re-test based on old discussions may also take place. In all observed situations, there are occurrences of continual doubt as to whether it has been solved or not. In the end, once the doubt is cast over the problem solution over a long period of time, unless concrete test cases have been provided as fully encapsulating the problem, the doubt remains.

2.5.7 Project Survival

Another type of IIC is to assess community health. Project survival occurred as the first instance of which theoretical sampling led our work to raise the conceptual level. We observed little to no concern for project survival in Lucene. We later observed, upon coding, that Lucene’s manifestation of survival occurred during its migration from SourceForge to Jakarta Apache. The chance need in this case was constant by the community. Even so, external chance need were still processed. Also, during project infrastructure migration, IIC dominated the process until the piping problem migration was completely solved. In seeking more information to saturate our understanding of this type of IIC, we observed that not all projects survived past Jakarta retirement. We then observed the lack of IIC as a warning sign of project death:

SLIDE-DEV-1-C: I’m not following this list long enough to judge your candidate’s coding merits. But there has been a lot of silence around here lately, which is not typically a sign that there are potential committers lining up.

ALEXANDRIA-DEV-2-C: Nobody responds, hence the project is more than closed, it’s dead.

2.5.8 IIC as a Basic Social Process

Basic social processes are just one type of core category: all basic social process are core variables, but not all core variables are basic social processes. They mainly differ in that basic social process have two or more clear emergent stages [22, p. 96]. They give the feeling of
process, change and movement over time [22, p. 97]. We already discussed IIC and its meaning. We now extend its notion as a basic social process, that is, we highlight how it manifests itself in different stages. Our understanding is that IIC is manifested as a way to achieve information gain. We use figure 5 as reference to illustrate its stages. In chance need, IIC is still undefined, but it is manifested as a service. As previously mentioned, a desire of continuous feedback may be desired from a company. While processing the piping problem, IIC is used as a tool for its solution. In particular, it affects the participants of IIC by increasing their information gain from suggestions to solve the piping problem. More specifically, a stage of participants own belief in the solution is evident, moving from an informal stage of seeking for consent, and optionally a formal stage of consent (voting) in case no agreement is reached or many solutions are provided, making it hard to reason over a conclusion. IIC may trigger iteration blockers, where a solution is never reached. It may then move to a stage of changing means of communication in different communication channels. For instance, the iteration blocker may suggest that the discussion continues somewhere else or be reopened in personal conversations, mailing lists, JIRA, etc. Time weakens IIC in that the longer it takes to solve by having longer discussions or taking long time to iterate, it leads to problem uncertainty and eventually not being solved. At such point, piping problems are only "resurrected" many months later or even years. After this stage, IIC now moves to a stage where it used to assess the status of the problem (for instance, knowing what ever happened to the results). Finally, over many problems death, IIC may move to a stage of assessing project health. In such instances, IIC may be used to assess if there is still any interest on the project or what action to take.

2.6 Evaluation

A initial concern of this work was in regards to the need of an expert. Glaser discusses this matter in Minus Mentoring [23, p. 5], which argues that a researcher should be able to conduct the method based on the readings of his books. We also discussed our interpretation in section 2.2 so others may assess it.

In following the method recommendations, we decided on subdividing the work as follows: The author, who is an undergraduate and has the least substantive knowledge on the field would learn and apply the method. The advisors of the author, would then evaluate the results from a software engineering perspective. The first advisor would follow in more closely with discussions when the first find it necessary, and would also discuss the method itself. Discussions were not to request information (this would bias the analysis) over the substantive area, but to assess over saturated categories if they make any
sense at all. This distinction is a very important point as argued in [23, p. 70] and can be seen as a derailing grab. That is, the author can be derailed from the grounded theory work because the not yet finished results cannot be coherent in conformance with other work. Thus, by making sense here we mean in both conformance to the GT method and in a consistent way, that is not trying to compare to the existing body of work in software engineering in seek of conformance. Finally, the second advisor who had not seen any step of the method nor the results at all would assess its contributions. Glaser [23, p. 10] also discuss the matter of collaboration in that collaborators should not write together. As conflicts arose, long discussions were followed in order to better understand the phenomena. This led to new properties and more data sampling. We were also careful to avoid biasing the data just to avoid discussions.

Unlike most qualitative and quantitative methods, GT suggests a process for evaluation of the methodology:

The credibility of the theory should be won by its integration, relevance and workability, not by illustration used as if it were proof. The assumption of the reader, he should be advised, is that all concepts are grounded that this massive grounding effort could not be shown in a writing. Also that as grounded they are not proven: they are only suggested. The theory is an integrated set of hypotheses, not of findings. Proofs are not the point. Illustrations are only to establish imagery and understanding as vividly as possible when needed. It is not incumbent upon the analyst to provide the reader with description or information as to how each hypothesis was reached. Stating the method in the beginning or appendix is sufficient, perhaps, with an example of how one went about grounding a code and an hypothesis [22, p. 134].

In sections 2.3 and 2.2.1 we have already discussed in length our interpretation of Classic GT.

In Glaser and Strauss [25, p. 97], an accounting scheme is also provided to assess if the comparative method was properly followed. Although this is not required to validate the use of the methodology, since the method in its full extension is still new in our field, we highlight them here.

- Is the author’s main emphasis upon verifying or generating theory?
- Is he more interested in substantive or formal theory?
- What is the scope of theory used in the publication?
- To what degree is theory grounded?
• How dense in conceptual detail is the theory?
• What kinds of data are used, and in, what capacity, in relation to the theory?
• To what degree is the theory integrated?
• How much clarity does the author reveal about the type of the theory he uses?

2.7 CONCLUSIONS

In this work we have applied Classic GT in OSS. We started the analysis without any assumptions, seeking to observe what was the main concern of the project, and how it was processed from the data. We concluded that IIC is a relevant latent behavior used to solve project concerns. We noted that such concerns occur by a Chance Need, which are stochastic due to the unknown motivation of the generated Problems. Furthermore, we noted that IIC can be seen as a service by an external party from the community, which in turn contributes to the sustainability of the community by earning contributions from IIC. Finally, we noted that IIC is a process that occurs even upon project death. We have also provided more insight on how the method can be conducted using assisting tools, and highlighted some of the definitions from a software engineering perspective, which are still few in our field.

For future work, we are interested in using the learned concepts to create a model. In Glaser [22, p. 62], it is argued that both factor analysis and grounded theory derive from a concept-indicator model. The GT concept-indicator model is based on constant comparison of indicators and the generated concepts among themselves as they are being generated, while [25, p. 63] factor analysis is a summing procedure where the meaning derives from indicators by their total interrelations and clustering into meanings. Instead of applying factor analysis, we are more interested in applying structural equation modeling [40], which extends factor analysis and allows for regression models to be applied over constructors. Our final goal would then to verify our observed GT model against other models created from existing literature.

Concretely, the GT work led us to the following conclusions:

• Piping problem processing is a main concern in OSS which is processed by IIC.

• IIC diminishes risks of applying a bad quality solution. In fact, this IIC characteristics is what, in our belief, makes it a good (service) for interested individuals in using the OSS solution.
• Different communication channels are used for solving the problem. This leads to different kinds of IIC and associated solution aids. Patches, which are usually seen as the solution to solving issues, consists of only the later phase of problem solving. Observing the evolution of problem solving might better capture how effort is distributed in problem solving.

• Different kinds of problems occur, mostly logical and technical. Logical problems relate to external users understanding. Documentation and code formatting might reduce effort spent on issues may be seen technical problems (e.g. bugs in code).

• Discussion is a major tool in OSS for dealing with many types of problems. Being able to capture what is the type of problem (logical or technical), and the communication channel might help in assessing amount of effort.

• Solution Aids are iteratively given during IIC, which culminates in more discussion. In this sense, commits, tests and other artifacts provided cause more discussion to occur. On the other hand, discussion also cause more artifacts to be suggested (patches being only a type of them), as an iterative process. We thus believe theoretically that there exist a relation between both, but such relation is only partial.

Being a core category, we decided that discussion is an important variable to be measured in assessing effort in OSS. Being related to the core category, support aids are important to be measured. From the next chapter and onwards, we will consider them as actions that are performed in order to address a given problem. Lastly, the problems, being logical or technical are related to code, and as such measures that account for their modification are important, which we decided to use churn.

We will define more formally each of those proxy effort estimators in chapter 4, after we discuss how the project data has been assembled.

2.8 Chapter summary

In this chapter, we conducted in OSS the first phase of our framework, and obtained a rich abstraction of potential effort estimators and how they relate to other phenomena within OSS.

In order to guarantee validation of our results, we first described our interpretation of GT, the required phases and evaluation methods, and applied it to OSS.

As results, we obtained as proxy effort estimators discussion, actions and churn, which are associated to the core category and related to it.
The next chapter presents a literature review of related work to each phase of our framework, and therefore was used to support our decisions on this and the upcoming chapters.
In despite of the many studies and systematic reviews labelled as software effort estimation models, we have found, however, that most of conducted work have been related to software development effort estimation. Despite the scope of this work being concerned only with software maintenance effort models, we discuss in this section related work of both sub-fields since both analysis and data understanding overlaps. In particular, the sub-field of software maintenance data can be considered a finer grain of the software development data, since we explore metrics at the file level instead of the project level.

In the upcoming sections we discuss important related work of both sub-fields that spans across all available years over the selected conferences and journals. Specifically, we reviewed all years of Transactions on Software Engineering (TSE), Transactions on Software Engineering and Methodology (TOSEM), Journal of Software: Evolution and Process (SMR), International Conference on Software Maintenance (ICSM), Conference on Software Maintenance and Reengineering (CSMR), Predictor Models in Software Engineering (PROMISE), Mining Software Repositories (MSR), Working Conference on Reverse Engineering (WCRE), International Conference on Software Engineering (ICSE), Empirical Software Engineering and Measurement (ESEM), Symposium on Applied Computing (SAC) (see table 1). The discussion is organized by subjects that relate to our work.

In order to summarize the body of knowledge acquired in all venues, after informally seeking for related work to ours, we were able to formalize and answer the following questions. The first three questions are important for deciding which learning methods can be employed on the last phase of our framework, prediction (chapter 5). The forth question gave us basis in considering threats when acquiring data in chapter 4. The last question was used to motivate our use of GT in chapter 2 for a metric mapping to one of the GT concepts.
Table 1: Literature review symposiums, conferences and journals where the literature review was conducted.

<table>
<thead>
<tr>
<th>Name</th>
<th>Years</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSE</td>
<td>1976 - 2013</td>
</tr>
<tr>
<td>TOSEM</td>
<td>1992 - 2013</td>
</tr>
<tr>
<td>SMR</td>
<td>1996 - 2013</td>
</tr>
<tr>
<td>ICSM</td>
<td>1993 - 2012</td>
</tr>
<tr>
<td>CSMR</td>
<td>1997 - 2013</td>
</tr>
<tr>
<td>PROMISE</td>
<td>2009 - 2012</td>
</tr>
<tr>
<td>MSR</td>
<td>2004 - 2013</td>
</tr>
<tr>
<td>WCRE</td>
<td>1993 - 2012</td>
</tr>
<tr>
<td>ICSE</td>
<td>1976 - 2013</td>
</tr>
<tr>
<td>ESEM</td>
<td>2007 - 2012</td>
</tr>
<tr>
<td>SAC</td>
<td>1993 - 2013</td>
</tr>
</tbody>
</table>

What are the most popular machine learning effort models? The most popular machine learning techniques were case-based reasoning, artificial neural networks, decision trees, bayesian networks, support vector regression, genetic algorithms, genetic programming, and association rules.

What are some of the statistical effort models? Ordinary Least Square (OLS), OLS regression with log transformation, OLS regression with Box Cox (BC) transformation, robust regression, ridge regression, least median square regression, MARS, CART, model tree, multilayered perceptron neural network, radial basis function networks, case based reasoning, least square support vector machine.

Are there best effort prediction models? No. Related work has claim that this can be due to the used accuracy metrics to compare models. However, recent work has found that an ensemble of the best methods for different datasets can surpass all individual methods.

Are there any concerns with data quality? Yes, but very few. A whole thesis [9] has been devoted on the subject, having the issues to file mapping being the most important threat in the reported work.

When effort time in hours is not available, what effort proxies are used? Churn had the highest occurrence in related work.

The following subsections discuss in more details how each question is answered and also provide discussion of other secondary related work to ours.
3.0.1 Software Development Effort Estimation

There are several approaches to model software development effort. Prediction models based on data vary mostly between statistical and machine learning approaches. According to Wen et al. systematic review [78], the most popular machine learning techniques were case-based reasoning, artificial neural networks, decision trees, bayesian networks, support vector regression, genetic algorithms, genetic programming, and association rules.

Some of the statistical models on the literature use ordinary least square regression (OLS). They include OLS regression with log transformation, OLS regression with Box Cox (BC) transformation, robust regression, ridge regression, least median square regression, MARS, CART, model tree, multilayered perceptron neural network, radial basis function networks, case based reasoning, least square support vector machines. [19]. Other less frequently used models include demi regression [54].

Claims in respect to which algorithm outperforms others were inconsistent across studies. Very recent studies have shown that this inconsistency may be due to the measures employed for model comparison such as mean magnitude of relative error [53], in despite of their popularity as accuracy metrics together with mdMRE (median magnitude of relative error). Other methods for model validation employed varied across holdout, leave-one-out cross-validation, and n-fold cross-validation. [78]. It was found that although not a single prediction model may work, an ensemble of the best prediction models can successfully outperform individual prediction methods [43].

Another kind of well known used method for effort estimation is analogy based effort estimation. In analogy based effort estimation, effort estimates are generated for a test project by finding similar completed software projects. Each row is given by a project, and the effort and remaining variables in columns. Project similarity is then assessed based on euclidian distance, resulting in clusters of similar projects. Different variations of the described method are suggested in the literature [41], [37].

Although not an automated model of prediction, expert judgment is still a subject of interest in the literature such as effort-irrelevant variables influence in effort prediction [32], and impact of lesson-learned sections in effort estimation [33].

Hypothesized effort predictors variables include project size, development, and environment related attributes [19]; budget and schedule pressure [59]; impact of maturity level as specified by CMMI [7]; functional complexity [45]. Furthermore, code size features have been found to be important in effort prediction [42]. Other means of predicting development effort has also been proposed by Robiolo et al.
Jørgensen and Shepperd [35] conducted a systematic review of software development cost estimation studies. They found in 10 journals from 1989 to 2004, most research interest in the cost estimation field was concerned with evaluation of estimation methods (61%), followed by size measures (20%) and organization issues (16%). The top estimation approaches consisted of regression (49%), function points (22%), expert judgement (15%), theory (13%), analogy (10%), the other methods all occurring less than 10%. They also called attention that they could not find any study that had an in-depth data collection and analysis of how estimation methods were actually applied on real life evaluation works for estimation methods. Other cost estimation studies have been related to uncertainty in software cost and its impact on optimal software release time [80]. In another systematic review by Kitchenham, results were inconclusive respect to the viability of cross-company estimation models [39].

### Dataset Concerns

Bad data can ruin any analysis. "Garbage in, garbage out" is as true today in this era of terabytes of data and distributed computing as it was in 1954 when How to Lie with Statistics was published ([31],[76]. In [47], Liebchen and Shepperd, after conducting a systematic review of available empirical software engineering studies, concluded that the community needs to consider the quality and appropriateness of the data set being utilized and that more research is needed to identify and repair noise. They also argue in favor of using sensitivity analysis to assess conclusion stability with respect to the assumptions that must be made concerning noise levels.

In Bachman PhD Thesis [9] and associated published works [10], [12], [15], [14] many domain related data quality issues are brought to light that directly affect the conclusions that are drawn from mining software repositories. The most important remark is in respect to the issue to file mapping threats, that is, given that not all code files are systematically mapped to issues, the analyzed mapped files may be biased towards a small population of files that may not be representative of the entire project.

In despite of Liebchen and Shepperd concerns with data quality issues, to the best of our knowledge, Bachman et al. works [9] contains the most complete list of warnings in conducting analysis on mined software engineering datasets. Aside from it, we have only found concerns with the used dataset were related to low amount of data points, and purpose of study related to the assembled data [19]. In our work in chapter 4 on threats to validity (4.3) we extend the list of concerns...
considerable, having obtained them during both data exploration and conducting GT.

Mair et al.\cite{50} performed an exhaustive search from 1980 and onwards in three software engineering journals (Information & Software Technology, Journal of Systems & Software, and IEEE Transactions on Software Engineering) for research papers that used project datasets to compare cost prediction systems. They concluded that quality, appropriateness and suggested assessment of the way results are presented to facilitate meta-analysis. Also, only 70% are available on public domain, and over a quarter were not easily available. Available start and end date of the dataset only occurred in 21 out of 74 datasets, and datasets varied considerable in size and amount of available variables. About a quarter of the datasets did not report whether the available projects within the dataset were from a single company or multiple companies. Almost 20% of the datasets also do not contain country of origin. The majority of datasets were reported to be used only once, and the most used ones were obsolete. All together, these concerns are important when drawing conclusions from different projects.

In our manual inspection, we have not found among all datasets reported on the discussed papers of this chapter any dataset available on a release basis level. Due to this, we used the dataset that motivated this work \cite{75}, which is presented on the next chapter 4.

3.0.3 Software Maintenance Effort Estimation

In so far we have considered only software development models, this section discuss maintenance effort models, which is the focus of this work. Maintenance cost estimation, which is our research goal, not as widely covered as the software development cost estimation field. They may require identification of meaningful factors of maintenance cost and model prediction models combination of different areas within a company \cite{17}.

Analysis of maintenance effort models includes correlation analysis and linear regression models \cite{17}, principal component analysis with regression \cite{61}.

Previous studies in maintenance effort prediction (given in hours) using file metrics used module level as number of modifications, subsystem changes \cite{66}; number of applications, number of modules and total code size \cite{64}; software components, lines of code, McCabe cyclomatic complexity, number of control variables, halstead software science volume and number of logical branches not used \cite{49}.

We have also found that models for estimating software corrective maintenance effort have to be customized based on the defect profiles and cost drivers of each company and project to be useful \cite{46}. Other similar models towards maintenance effort prediction that are not on a file level...
and have already been cited includes [55], [27], [16]; use cases [44], function points [60], [8], build maintenance effort [52], comprehension effort during maintenance [62].

To the best of our knowledge, and according to related work [61], [29] the only work that has considered analysis over release level of a project is by Basilli et al. which dates to ICSM 1998.

3.0.4 Effort Proxies

Given the importance of this section to our work, we discuss in more details the related works that investigated effort proxies.

Karus and Dumas [36] studied the problem of predicting the coding effort for a subsequent year of development by analyzing metrics extracted from project repositories, with an emphasis on projects containing XML code. Thirteen open source projects were used. They used machine learning algorithms to generate models to predict one-year coding effort, measured in terms of lines of code added, modified and deleted. The study also shows that models trained on one project are unreliable at estimating effort in other projects. It is also pointed out that relatively little attention has been given to estimating coding effort based on organizational metrics extracted from version control systems, such as developers’ activity, therefore leading the author to conclude that a wealth of information available in version control systems - beyond the code itself - that can be used to predict code churn.

Hayes et al. [29] derive a model for estimating adaptive software maintenance effort in person hours. They used regression models, and found that a number of metrics such as lines of code changed and number of operators changed were found to be strongly correlated to maintenance effort. They hypothesize that the maintenance effort for a software application depends on measurable metrics that can be derived from the software development process. The results suggest that percentage of operators changed and the number of lines of codes changed edited, added or deleted (DLOC) were the most effective for predicting adaptive maintenance effort. The used data was not available, and the amount of data points is arguably very low (9, 10, 8 and 6 on each of the 4 used projects).

Basten and Mellis [13] conducted a survey (web-based questionnaire) to gather data on the current situation of software development effort estimation due to participants perception being in disagreement with objective assessment, they concluded that research needs to find new measures to assess the actual estimation accuracy.

Sjoberg et al. [73] investigate whether suggested maintenance metrics estimators are consistent among themselves and the extent to which they predict maintenance effort (expressed as the maintainability index metric) at the entire system level. They found that the met-
metrics were not mutually consistent, having only system size and low cohesion were strongly associated with increased maintenance effort.

We have found works using the number of days elapsed between to releases of a given project \([1]\).

3.0.5 File Metrics

In regard to used file metrics, we have observed from the previous sections that effort related work related to either software development or maintenance effort or cost estimation models do not consistently employ file metrics. Furthermore, we have not observed any concern in deciding which file metrics were more appropriated. Such decision may be done either empirically or based on literature review. We have found only a single work that does so empirically, by using a multilinear regression to validate the set of metrics \([21]\). In our work, we decided our file metrics based on literature review. An extensive discussion and the rationale behind the choice has already been made in the study that motivated work, and therefore we do not discuss them here \([75]\).

3.1 Chapter Summary

In this chapter we inquired how related work could support the different parts of our framework. We were able to distinguish between maintenance and effort prediction related works, and the need of more flexible effort estimators. We also found there are not an agreement over effort estimators, aside from churn which have appeared more often as related to effort in hours. Different data quality concerns such as the issue to file mapping and different granularities of analysis motivated our decision to remain on a release level grain, and manually investigating different threats to validity, not only on the chosen statistical and machine learning models, but also within the data, as is discussed in chapter \(4\).

In the next chapter, we use the learned lessons from the literature review to organize and motivate the different types of conducted analysis.
Part II

DATA COLLECTION AND DATA STORAGE

Before drawing any statistical analysis towards our problem, we must decide how to abstract irrelevant aspects of the phenomena while capturing those of interest. Data collection and data storage may therefore limit the possibilities of analysis in later phases. In this part, we discuss how we collected and prepared our dataset based on the previous part decided maintenance effort estimators and structural complexity file metrics. From the obtained results, we present the tables of Derby, Lucene, Pdfbox and Ivy that are used subsequently for quantitative analysis.
4

SOFTWARE ENGINEERING DATASET REVIEW

You used to be much more...“muchier.”
You’ve lost your muchness.

— The Mad Hatter - Aline in Wonderland - Lewis Carroll

4.1 INTRODUCTION

In this chapter we present a summary of how the data is available, collected and assembled in Derby, Lucene, Pdfbox and Ivy project tables. Specifically, we highlight the major steps on moving from the data raw form to just before the analysis in chapter 5. We explain the need of certain transformations, and assumptions that were required and how they lead to threats to validity of our work in combination to what we have observed during GT in chapter 2. A more technical discussion can be found on the thesis that motivated our work [75, ch.4, ch.5].

In order to provide means to validate our results, we made available a comprehensive report of each transformation and its rationale from the mined data stored in our database to the results presented here. They are available as R scripts online¹ for the associated chapter.

4.2 RAW DATA

In order to explain the raw data, we first explain the main process in which maintenance is conducted in OSS.

In OSS people contribute online by means of varying tools. This is highlighted in (1) in figure 6. The usual infrastructure consists of users and developers mailing lists, an issue tracker to register bugs, new features, etc such as JIRA and a code repository (e.g. GIT or SVN). Other tools may be used such as IRC for communication on real time, websites that provide project information, wikis and so on (other tools are indicated as other data in figure 6).

¹ https://github.com/carlosandrade/undergraduate_final_project
Before moving to data collection and how data from different sources are connected (2 to 6) in figure 6, we explain how (1) occurs in more details.

In order for people to contribute to a project in Apache Software Foundation, an issue is first opened. From GT, we observed that it can be the case that some discussion occurs in mailing lists before an issue is created.

After an issue is created, the creator and other people can contribute to it. Contribution occurs by means of discussion or by means of patches. We have already discussed how discussion contribution occurs in GT in chapter 2. Figure 7 shows how patch contribution occurs. (1) A patch is first attached to an issue in JIRA, which has a page on
the web. (2) **IIC** then takes place, which (3) may lead to more patch submissions by the same or other contributors.

The last step is a commit as shown in figure 8. (1) A patch contribution that is regarded as a viable solution by **IIC** is taken by a comitter from the issue page. (2) An acknowledge of this is made by means of **IIC** so that either is consent that the solution is viable by the committer, having then the patch contribution applied in the code repository. In other words, zero or more of all of the patches contributions that are made to an issue find their way to the code repository, and those which do so are called commits. Such submissions to code repository can only be done by comitters as the name indicates.

We now move our attention to the mining steps of figure 6 (2 to 6). The link between step (1) and (2 to 6) occurs by means of the activities that are logged of the just described process of contributions. The description of the collected information from the log occurs in two different ways for the maintenance effort estimators and the structural complexity file metrics. For the maintenance effort estimators:

**Discussion:** For each issue, we collect how many comments were posted until the issue was set as closed.

**Actions:** For each issue, we identify all submitted patches and which of them were turned into commits. After this identification, we analyze which files were contained within the patches, and as thus which files were or not part of commits. The end result is a list of files that were submitted one or more times to address an issue, some of which have been turned into a commit. We count how many times each of those files occur per issue. That is, in how many patches a given file occurred for a given issue. The number of occurrences is named as actions, since we count the file when it was submitted as a patch and when it has been turned into a commit. Recall from figure 8 that a comitter is a different person who may apply some changes to the file when taking it as a commit, and thus we should count file occurrence either when it occur as a patch or as a commit. An exception to this workflow is when a commit is directly made without being a patch, which is counted as well.
CHURN: Churn is the amount of lines added and removed from each file when it is modified. In the defined process of figures 7 and 8, a file is modified whenever it occurs within a patch or commit. We use the same procedure as in calculating actions, in that we list all the files that occurred within for each issue, either as part of a patch or as a commit. The extra step to calculate churn is to link all those files to their revision in the code repository. Once this is done, we are able to obtain a diff of lines added and removed, thus obtaining the file churn. In other words, churn is obtained using the code repository. This code repository dependency will be important for an important assumption made in our work as we will discuss later in the threats to validity section (4.3).

For the structural complexity file metrics we download from the tags repository the project releases and use CKJM2 to calculate the file metrics. We are also aware that CKJM file metrics provide simplifications of some of its calculated metrics. We discuss this also in the threats to validity section (4.3).

It is important to note at this point that both actions and churn are file associated measures, and therefore are calculated on a file change grain level. Structural file complexity on the other hand is measured on a release level. Intuitively, it would be expected to conduct all the analysis on the lowest grain possible, which is file change, however, as shown in our literature review in chapter 3, most projects have conducted analysis even on a higher level of granularity (project level). The reason for this is the difficulty to apply each change made to a file and apply file metrics to it.

Furthermore, the available log imposes restrictions on the grain of the data that will be obtained, as well as the process of the community, such as the one described for Apache Software Foundation. For instance, have JIRA provide means to log to which issue a discussion belongs, and having a process (as unnatural as this may seen) have people indicate to which issue they would be discussing, then we could get discussion on a file change level without applying any heuristic (we discuss possible heuristics of data transformation across grains in the following analysis chapter 5).

From an analysis perspective, different grains may lead to repetition of values during the analysis. For instance, if 3 files are part of a patch of issue A who had 10 comments by the time it was closed, then each of those 3 values will be assigned 10 comments as their discussion value since there is no way to map the issue discussion to files. This is clearly a threat to validity.

In order to conduct analysis in this data, we need to define the final grain in which a data point (row in a table) will contain. That

2 http://www.spinellis.gr/sw/ckjm/doc/metric.html
Table 2: Partial data table containing all maintenance effort estimators and two of the file metrics of CKJM. File path, issue code, release number and file metrics are omitted due to space, but uniquely identify each row of this table.

<table>
<thead>
<tr>
<th>row</th>
<th>churn</th>
<th>actions</th>
<th>discussion</th>
<th>raw_loc</th>
<th>ckjm_dit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>126</td>
<td>1</td>
<td>14</td>
<td>128</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>1</td>
<td>3</td>
<td>324</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>59</td>
<td>1</td>
<td>0</td>
<td>342</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>482</td>
<td>3</td>
<td>0</td>
<td>573</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>34</td>
<td>1</td>
<td>0</td>
<td>573</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>44</td>
<td>2</td>
<td>4</td>
<td>195</td>
<td>1</td>
</tr>
</tbody>
</table>

is, what uniquely identify each row in a table. We define the grain of our dataset to be a file which was submitted to an issue in a given release. Table 2 shows a partial table containing all our maintenance effort estimators and two of the file metrics from CKJM. We create one table of this format for Derby, Lucene, Pdfbox and Ivy which is then used for analysis in chapter 5.

Lastly, we discuss briefly figure 6 (2 to 6) links. In order to connect in table 2 all the metrics together in our defined grain, (3) we had to find a way to connect source code revisions to issues. This is done by using pattern matching over commit labels while seeking for issue codes. Once a match occurs, all files are associated to the given issue.

Since file metrics are applied directly by downloading code from each tag folder of a project, this data can be seen as "other data" in figure 6. (2) In order to link the calculated file metrics to code, we use the relative file path of the downloaded tag folder, and map it to the repository revision file path, by means of a regular expression. This poses a challenge on extracting data from new projects, as this regular expression often have slight differences across projects, in despite of all being part of Apache Software Foundation, and having a trunk, tag and branch structure. Regular expression creation is also made manually, which is time consuming.

We do not (4) tie issues to mailing lists. This would be of interest in order to include discussion not only from issue comments, but as well exchanged mails belonging to issues. Identifying issues within an mailing list is considerable harder, since the data is not structured and there is no guarantee that the issue code being identified actually corresponds to a problem being attempted to solve.

Steps (5) and (6) are strictly technical, and as thus are not discussed here. As stated in the introduction of this chapter, the technical details of how this was done can be found in [75]. We only highlight specific
details here that directly affect the analysis that is conducted in this work or poses a threat to validity.

4.3 Threats to Validity

4.3.1 Mining Threats to Validity

As with any empirical study, our results are subject to a number of threats to validity. Below we discuss the most significant ones: the threats to validity caused by the choice of maintenance effort measures, the choice of projects to study, and the limitations of our data extraction methods.

We employed the three types of file-level effort measures because, as stated earlier, developers in open-source projects do not log their hours. Even in industrial settings, accurate effort logs are rare. Moreover, we need maintenance effort at the file level, which is even rarer. We hypothesize that our three measures—churns, actions, and discussions—can adequately approximate the maintenance effort spent on a file. This is, however, a hypothesis that is impossible to directly test given our existing data-set and has been done by means of the abstraction resulted from GT.

In our calculation of discussions as a measure of effort, we did not mine the projects’ mailing lists. We avoided the mailing lists because it is difficult to accurately link backwards from a mail message to a specific file/release/issue combination. But this means that we are not able to accurately assess the full body of discussions associated with a project, and hence we are potentially biasing our discussions measure.

We considered several other possible maintenance effort proxies, such as the time used to resolve an issue, the number of discussions by email, or the number of defects associated with a file. We chose not to include these other possible proxy measures for various reasons: the inability to distinguish work time from slack time, our inability to reliably and automatically link email discussions with files, and the difficulty of attributing a bug fix to a specific release.

We have also assumed that all bugs/change requests that were resolved between one release and the next were attributable to the latest release. However, it is possible that a few of the bugs or change requests actually applied to an older release that is still being maintained. These bugs or change requests are described as backports by Bachmann et al. [11].

The effort proxies we proposed are based on the assumption that the project employs a version control and issue tracking system from which the effort proxies can be extracted. As a result, the proxies won’t be applicable for projects without comparable tools. More importantly, as with any research relying upon bug fix data, our study
needed to link files with issues by finding developer’s commit messages indicating which issue the commit is intend to address. As Bird et al. [14] have pointed out, in many projects, only a small portion of bug fixes are actually recorded in version control systems. Since our study compares the behavior of the three proxies using the same dataset, the extent to which our results are biased due to the incompleteness of the dataset needs to be evaluated.

Another threat to validity is that we only examined open-source projects. While we expect that similar results will be obtained from industry projects, this is an open research question. And within our chosen projects, we ignored files that had more than one class defined in them. However, this only excludes 7% of the files. We have only investigated 5 projects thus far, all of which were from the Apache foundation, and we only investigated a subset of the possible metrics that we could have considered. For example, we could have considered propagation cost metrics with different path lengths and different decay factors. A larger study employing more projects and more metric types would improve the validity of our conclusions.

There are many extraneous project factors such as the inherent difficulty of bugs and issues, and the inherent skill of the developers that add noise to the data. Further study that controls for some of these noise factors is thus called for. For example, we are already seeing some promising results by categorizing change requests into groups of small, medium, and large requests. This way, we can see what the influence of source code metrics is on the different sizes of changes, and eliminate the noise that the inherent variation in change complexity adds to the data.

4.3 Threats to Validity

4.3.2 Inherited Threats

4.3.2.1 Disabled Subversion-Jira Plugin

As of the start of July 2012, linkages between files and issues were no longer available for Apache Software Foundation projects due to problems with a plugin. This is important to note, because it was due to the plugin that developers used the convention of labeling commits with the JIRA issue being addressed. Subsequent to this event it is important to verify if the balance of issues and labeled commits do not affect the data represented in our dataset. An investigation of the proportion of commits with and without an identifying issue can be an important criteria on selecting projects. For example, a project where it is observed that a significant portion of commits do not refer to an issue can bias the model since the characteristics of those files won’t be used in the training set. Despite the plugin threat, it is not a rule that every commit must have a labeled issue, so the analysis being conducted using our dataset should explicitly account for this.
4.3.2.2 Outliers

Well structured commits should be easier to digest by reviewers and thus find their way to the main trunk branch more quickly. An outlier in this case would be: (1) a commit with large quantities of code added and removed before going into the main trunk branch, and (2) well structured commits that take much longer time compared with others.

We analyzed outliers by observing the min and max values of each predictor (file metrics). We also compared the churn rate - \((\text{loc}_{\text{added}} + \text{loc}_{\text{deleted}})/n_{\text{commits}}\) - to the time to solve an issue on each project. The churn rate highlights any outlier values in churn (too large or too small); by examining this versus time we identify issues with low amount of code changed over many commits in a brief time period.

4.3.2.3 Project Migration and Inherited Threats

When using our dataset, the consumer should be aware of migration of issues from other systems that might have occurred. For instance, Lucene has many JIRA issues that were migrated from its old Bugzilla system (used when it was part of the Jakarta Project). Such issues are indicated by a link (e.g. see Lucene-1) pointing to the original Bugzilla issue (which is no longer available, making referencing among issues by following the discussion harder). Another case occurs in PDFBox (e.g. see pdfbox-99) where the issue was originally moved from SourceForge. In observing the SourceForge migration, one may see that calculating the amount of discussion can be misleading since the migrated comments were integrated as part of the issue description instead of being added as distinct comments in Jira. Detecting such a migration however can be done by identifying the label on the issue description ‘[imported from SourceForge]’.

4.3.2.4 Completeness of Migration

In considering the study of a project the consumer should be aware that our dataset does not include the early days of the projects. Most projects have migrated over a different infrastructure for code version control, bug tracking and mailing lists, and tracking all of this would require much more extensive tooling.

4.3.2.5 Project Culture

Projects have different cultures and these cultures normally evolve over time. This may affect the kinds of project artifacts created and how they are used by project members. For example, we have identified that in Lucene some issues have ‘follow ups’ conducted over the mailing list (rather than in their JIRA archive). Different projects
might employ different criteria for deciding where such ‘follow ups’ are housed. An example of such an evolution may be seen in Lucene’s mailing list where, over time, the e-mails evolved from consisting primarily of discussions among contributors to consisting primarily of JIRA notifications. Such idiosyncratic patterns of evolution could be a threat to validity in our dataset; this is currently an area of investigation.

This list of threats to validity is not exhaustive, but serves as an introduction to anyone considering usage of our dataset. A more complete list of, and analysis of, the threats to validity of this dataset is part of our future work. As we extend the dataset with additional projects we fully expect that additional threats will be manifested.

4.4 CHAPTER SUMMARY

In this chapter we have presented how our dataset was assembled, and based on the lessons learned from literature review 3, what different combinations of aggregation could be performed to obtain insight from the characteristics of the collected data. We have also presented an extensive list of threats to validity which were observed both during CT (chapter 2) and from the related work. Specifically, we were able not only to consider quantitative threats such as different granularities and file to issue mapping, but also inherited threats, which to the best of our knowledge were not addressed before in related work in this depth of detail.

In the next chapter we move to the last phase of our framework. We use the defined dataset from this chapter to build our prediction models and make our conclusions in predicting effort indirectly from structural complexity metrics.
In this part we discuss how the obtained project tables are further pre-processed for analysis. We conduct exploratory, correlation and prediction analysis while giving focus to how the process could be used from a stakeholder perspective.
In this chapter we discuss the analysis model given the explanation of the variables previous presented. In order to provide means to validate our results, we made available a comprehensive report of each transformation and its rationale from the mined data stored in our database to the results presented here. They are available as R and MATLAB scripts online\footnote{https://github.com/carlosandrade/undergraduate_final_project} for the associated chapter.

This chapter is sub-divided as follows: First we explain the pre-processing (5.1) steps we applied in the raw dataset explained in the previous chapter 4 and how it affects the upcoming analysis. Second, we explore the discussion maintenance effort estimator (5.2), which is the quantification of the GT core category IIC presented in 2. Third, we present the many different correlation analysis (5.3) we performed with our maintenance effort estimators and the measured structural complexity file metrics. Forth and last, we show our prediction analysis results by using the file metrics and effort estimators to predict discussion (5.4).

## 5.1 Pre-processing

Since our data is organized per project per release, as explained in chapter 4, we must inspect each project release before building prediction models or performing correlation analysis. One initial question is the amount of releases per project, and the amount of data points per release. As previously explained, a data point is uniquely identified by \textit{(file relative path, issue code, release number)}. Since we analyze each release separately, within a release each data point is identified by \textit{(file relative path, issue code)}. In order words, each release data point is a file who have been modified (as part of a path or commit) to fix an issue in that release.
Table 3: Derby Releases and associated amount of data points before pre-processing.

<table>
<thead>
<tr>
<th>Release ID</th>
<th>release</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10.1.1.0</td>
<td>97</td>
</tr>
<tr>
<td>2</td>
<td>10.1.2.1</td>
<td>162</td>
</tr>
<tr>
<td>3</td>
<td>10.1.3.1</td>
<td>57</td>
</tr>
<tr>
<td>4</td>
<td>10.2.1.6</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>10.2.2.0</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>10.3.1.4</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>10.3.2.1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>10.3.3.0</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>10.4.2.0</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>10.5.1.1</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>10.5.3.0</td>
<td>106</td>
</tr>
<tr>
<td>12</td>
<td>10.6.1.0</td>
<td>84</td>
</tr>
<tr>
<td>13</td>
<td>10.6.2.1</td>
<td>30</td>
</tr>
<tr>
<td>14</td>
<td>10.7.1.1</td>
<td>83</td>
</tr>
</tbody>
</table>

Tables 3, 4, 5, 6 shows for each project how many releases and how many data points (file relative path, issue code) associated to each release a project has before the pre-processing step. We can observe across the tables that there are many releases whose data points do not suffice for conducting neither correlation or prediction analysis. We filtered each table so that each release contains a minimum of at least 40 data points. This low amount of data points can be caused by any of the threats to validity considered in previous chapter 4, specially the file to issue mapping threat and our mapping from code repository and tags folders to their respective releases. Tables 7, 8, 9, 10 shows the final tables that are used for the exploratory (5.2) and correlation analysis (5.3). A substantial amount of releases have been removed from our analysis due to this filter. While for both exploratory and correlation analysis we can still examine the data distribution per release, we will be unable to use Lucene and Ivy projects for our prediction analysis (5.4) since they are left with only 1 and 2 releases respectively.

5.2 Exploratory Analysis

Figures 9, 10, 11, 12 shows the histograms across the filtered releases. In all project histograms, we can notice that the discussion distribu-
### Table 4: Lucene Releases and associated amount of data points before preprocessing.

<table>
<thead>
<tr>
<th>Release ID</th>
<th>release</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.9.1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2.2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2.3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>2.3.1</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>2.3.2</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>2.4</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>2.9</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>2.9.1</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>2.9.2</td>
<td>56</td>
</tr>
<tr>
<td>10</td>
<td>2.9.3</td>
<td>35</td>
</tr>
<tr>
<td>11</td>
<td>2.9.4</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>3.0</td>
<td>17</td>
</tr>
</tbody>
</table>

### Table 5: Pdfbox Releases and associated amount of data points before preprocessing.

<table>
<thead>
<tr>
<th>Release ID</th>
<th>release</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.1.0</td>
<td>43</td>
</tr>
<tr>
<td>2</td>
<td>1.2.1</td>
<td>48</td>
</tr>
<tr>
<td>3</td>
<td>1.3.1</td>
<td>23</td>
</tr>
<tr>
<td>4</td>
<td>1.4.0</td>
<td>56</td>
</tr>
<tr>
<td>5</td>
<td>1.5.0</td>
<td>53</td>
</tr>
<tr>
<td>6</td>
<td>1.6.0</td>
<td>10</td>
</tr>
<tr>
<td>Release ID</td>
<td>release</td>
<td>n</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>----</td>
</tr>
<tr>
<td>1</td>
<td>2.0</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>2.0-RC1</td>
<td>29</td>
</tr>
<tr>
<td>3</td>
<td>2.0-RC2</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>2.0.0-alpha-2</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>2.0.0-beta-1</td>
<td>25</td>
</tr>
<tr>
<td>6</td>
<td>2.0.0-beta-2</td>
<td>190</td>
</tr>
<tr>
<td>7</td>
<td>2.1.0</td>
<td>48</td>
</tr>
<tr>
<td>8</td>
<td>2.1.0-RC1</td>
<td>22</td>
</tr>
<tr>
<td>9</td>
<td>2.1.0-RC2</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>2.2.0</td>
<td>25</td>
</tr>
<tr>
<td>11</td>
<td>2.2.0-RC1</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 6: Ivy Releases and associated amount of data points before preprocessing.

<table>
<thead>
<tr>
<th>Release ID</th>
<th>release</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10.1.1.0</td>
<td>97</td>
</tr>
<tr>
<td>2</td>
<td>10.1.2.1</td>
<td>162</td>
</tr>
<tr>
<td>3</td>
<td>10.1.3.1</td>
<td>57</td>
</tr>
<tr>
<td>4</td>
<td>10.5.3.0</td>
<td>106</td>
</tr>
<tr>
<td>5</td>
<td>10.6.1.0</td>
<td>84</td>
</tr>
<tr>
<td>6</td>
<td>10.7.1.1</td>
<td>83</td>
</tr>
</tbody>
</table>

Table 7: Derby Releases and associated amount of data points after preprocessing.

<table>
<thead>
<tr>
<th>Release ID</th>
<th>release</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.9.2</td>
<td>56</td>
</tr>
</tbody>
</table>

Table 8: Lucene Releases and associated amount of data points after preprocessing.

<table>
<thead>
<tr>
<th>Release ID</th>
<th>release</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.1.0</td>
<td>43</td>
</tr>
<tr>
<td>2</td>
<td>1.2.1</td>
<td>48</td>
</tr>
<tr>
<td>3</td>
<td>1.4.0</td>
<td>56</td>
</tr>
<tr>
<td>4</td>
<td>1.5.0</td>
<td>53</td>
</tr>
</tbody>
</table>

Table 9: Pdfbox Releases and associated amount of data points after preprocessing.
Table 10: Ivy Releases and associated amount of data points after preprocessing.

<table>
<thead>
<tr>
<th>Release ID</th>
<th>release</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.0.0-beta-2</td>
<td>190</td>
</tr>
<tr>
<td>2</td>
<td>2.1.0</td>
<td>48</td>
</tr>
</tbody>
</table>

5.3 Correlation Analysis

We considered various ways of investigating the correlation of our cost estimators and file metrics. In chapter 4, we’ve shown how the data used in this chapter is assembled and the rationale behind it. We also pointed out that, file metrics are measured per release, while the maintenance effort estimators churn and actions are measured for each file change. Lastly, discussion is measured per issue. The net effect of mapping all these different grains to a single table where each row represents a (file path, issue code, release) will therefore cause values repetition.

Consider the two following examples, 4, as a motivation of performing multiple transformations for correlation analysis:

First, assume that an issue A has 10 comments when it was closed. This issue A can receive patches or commits to address it. Any patch or commit submitted to issue A will be assigned 10 comments for their discussion value. Given that either patch or commit may contain 2 or more files, all the files will be assigned the value 10 of discussion. Therefore, if in the end 10 files were submitted to issue A, all the 10 files will have assigned 10 as the amount of discussion.

Second, given that file metrics are measured only once per release, if this release contains 10 issues, the set of file metrics will be mapped
Figure 9: Discussion Histograms in Derby

to those 10 issues. If a given file B appears in 5 of these issues, he will have the same metric value being repeated 5 times.
These two examples show that both file metrics and the discussion values are repeated across files as already discussed in the threats section of chapter 4. For the correlation analysis, we considered applying the following transformations on the data to investigate if this repetition caused by the mapping would cause any significant impact over the correlations of both our cost estimators and our file metrics.

Concretely, we investigated the following transformations per project per release:
RAW: In "raw" form, we investigate the correlations as they were mapped by the transformations in our dataset discussed in chapter 4, that is, with the propagated repetitions.

RELATIVE: To calculate the relatives maintenance effort estimators and file metrics, we take the binary combinations of each release and its immediate predecessor release. We then generate the intersection of files on both releases. This is necessary since we can’t compare the relative of a variable value if doesn’t occur on both releases. We then divide the most recent release variable value by the previous release variable value of each file. For divisions by 0, we assign the value 0 for the intersection set. This transformation reinforces the following assumptions: First, the resulting correlations will refer to the increase or decrease rate of the maintenance effort estimator or file metric. Second, since we only consider the immediate previous release, this leads to a substantial decrease in the dataset size per release, in case the files is not reported in one of the two releases. Third and last, this transformation does not look on two or more previous releases. This because the transformation assumes that if the file does not occur in the previous release, it is because it has not been mapped to an issue, instead of not being modified. Under this assumption, looking more than the previous immediate release would then constitute a incorrect value, so instead it is discarded on the intersection.

RELATIVE PROPAGATION: Relative propagation is similar to the relative transformation, varying only on the intersection part. Instead of attempting to intersect with only the immediate previ-
ous release, this transformation visits all the previous releases available in attempt to find the file to calculate the intersect variable value. This difference thus emphasizes the assumption that the file was changed, and all issues have been properly mapped (since looking the previous releases that are not the immediate would calculate a wrong value if it wasn’t mapped). Together with the Relative transformation, we cover both assumptions, since there is no way to know if a file was not changed in the immediate previous release due to not being mapped, or if simple wasn’t changed at all in the previous immediate release.

**Aggregation:** In aggregation, we attempt to deal with the discussion and file metrics releases repetition from both provided examples. Specifically, we aggregate the maintenance effort estimators of actions and churn on the file change grain, and the discussion effort estimator, all to release level with varying aggregate functions. We calculate the mean, median, sum, max, and min for these 3 variables for the release level. This way, each file will be assigned only a single action, churn, discussion and file metric value for each release. Of course, this also leads to loss of information due to the aggregation transformation. The ideal scenario, which would be calculate the specific amount of discussion and change file metrics is not possible, as discussed in the threats of validity of chapter chapter 4. This leads to some loss of data points due to the aggregation.

**Relative Aggregation:** This is a combination of both the aggregation and the relative transformations. First the aggregation with all the file metrics are calculated, then the relatives are calculated. This leads to the least amount of data points, since data points are lost to aggregation and due to the relative transformations.

**Relative Propagation by Aggregation:** This is the combination of relative propagation and aggregation. Again, we first aggregate, and then we perform relative propagation, as previously discussed. This lead to loss of data points due to aggregation and relative propagation, although less than relative aggregation due to the propagation.

In this chapter we present only the raw transformation. Discussing each matrix would require many times more space than what is allowed for this work, and to a certain point unnecessary. To verify, we proposed a total of 6 transformations. Half of the transformations yields a 12 x 12 correlation matrix (file metrics + cost estimators versus themselves), and the other half of aggregation matrix is 24 x 24 (twice the size due to the varying aggregating transformations). This results in (144*3 + 576*3) cells to be analyzed (although the upper
and lower diagonal are the same, we end up looking them more than once, so this is a fair consideration). Lastly, this amount of cells is analyzed for each release, for a total of \((144*3^3 + 576*3)^13 = 28080\) cells. Furthermore, unfortunately, we did not notice any significant change over the transformations, suggesting that neither the repeated mapping values due to grain or way the variables are allocated justify the cost of applying such transformations for our given set of projects, releases and file metrics. The raw form is also the most easily obtained metrics, and in our case representative of the remaining transformations. We however report the correlation matrix in the online R script. Part of the transformations were also conducted on the thesis from where this work has built of.

Even considering only the raw transformations, a considerable amount of tables would still be required to be discussed (13 in total). Instead, we show the correlations as plots for each project in figures 13, 14, 15, 16. Each plot displays the mean rho value across all the project releases, and the associated rho standard deviation across all the releases. The correlation pair name also displays in how many releases has the correlation existed under a \(< 0.01\) significance level. Notice also that in cases where the pair is only reported once, the standard deviation is 0, and thus no "line" surrounds the correlation pair "dot". We show all correlation pairs but those of file metrics versus file metrics, since our main interest is of investigating the relation between maintenance effort cost estimators among themselves and file metrics. The manual analysis over the file metrics was conducted for investigating the potential use of the transformation in models that requires non-correlated predictors, as we will discuss in the poisson analysis in the prediction analysis section (5.4).

An interesting consequence of visualizing the rho values as aggregates and deviations instead of tables is that it is easy to observe which correlation pairs rho values are stable (does not vary much within a project or across projects), and which correlation rho pairs occurs only sparsely under a \(p < 0.01\) significance level across all projects. We can also easily notice who rho values are higher in comparison to others being reported.

From the presented plots, we observe that: Derby is the only project that shows correlations between maintenance effort cost estimators and file metrics, and do so mostly for a rho value less or equal than 0.4. Raw loc and ckm rfc file metrics correlates the most with actions, and only the churn maintenance effort cost estimator relates with discussion. Most of the file metrics lack a standard deviation line due to its occurrence in only one release of the derby project.

In the other projects, we mainly observe the occurrence between churn and discussion correlations. We can also observe that churn and discussion have a mean rho correlation always above 0.6, although not very stable given its standard deviation width across each
project. It is also important to note that given the amount of releases which remained for all projects but Derby being reasonable lower may also affect the fact that only Derby has file metrics being correlated to maintenance effort estimators.

In summary, from the correlation analysis, it appears that churn is the only way among our file metrics to relate to discussion, while raw loc and ckjm rfc are the only file metrics that relate to an maintenance effort cost estimator, which is actions.

In considering our variables definitions and their motivation under our hypothesis and choice of maintenance effort estimators and file metrics, we may also conclude that: Structural complexity relates mostly to the amount of changes required per issue a given file has. This amount of changes does not appear to relate to the amount of discussion. Instead, the amount of lines added and removed in a given file seems to relate to how much discussion is generated in a given issue.

Obviously, this is just a "rather optimistic" possible interpretation and insight over the results presented. A "less optimistic" interpretation would take into account the fact that the file metrics and actions only occurs in a single release out of 6 may also indicate that this only happened by chance for a single release, and the only expected relation should be of churn and discussion.

For the prediction analysis subsection 5.4, we take the "less optimistic" interpretation. Given that lower rho values of correlation do not necessarily lead to low prediction accuracy [71], we take into account all the file metrics and maintenance effort cost estimators in order to assess if it is possible to predict discussion.

5.4 PREDICTION ANALYSIS

As discussed in the correlation analysis subsection, we decided on attempting to verify if we can predict discussion from our other effort cost estimators and file metrics. Not only the correlation favors this attempt even under a pessimistic interpretation of the correlation analysis, but as well this was our initial framework goal: Identifying other proxy points of effort, which are not captured by the time variable. Even better, be able to predict them from other proxy points that has been reported in literature to relate with time.

Concretely, in verifying the correlation, it was clear that a significant portion of the main concern effort has been missing in looking only for time measures in OSS projects. We now assess if we can predict them, in despite of not having time logged for them.

For the prediction analysis, we opted for transforming the discussion variable in intervals. The motivation is due to two reasons: First, as we seen in the exploratory analysis section, the discussion distribution varies considerably in distribution across releases and has a high variation. Second, we believe that a predictor that reports well within
Figure 13: Derby Correlation Plot. Cost Estimators versus cost estimators, and cost estimators versus file metrics pairs are computed.

an interval rather than one that performs worse is better acceptable as a recommender system. Furthermore, in practice we would expect
that a person would intuitively reason about intervals of effort that are larger, average or low (or other granularity of intervals), rather than deciding if 5 comments is critical when 4 is not for discussion.

This lead is to the question of how to define the intervals and how to conduct the prediction analysis with them. First, we randomly\(^2\) choose one release from Derby and one from Pdfbox to be our test datasets (recall from the pre-processing section (5.1) that the remaining projects didn’t have enough releases for conducting such analysis). We then analyzed for the remaining releases of Derby and Pdfbox second and third quartiles for each release.

This follows the intuition of someone who is used to the project to reason about how much discussion usually happens over releases.

\(^2\)http://www.random.org
and then try to infer if the amount of discussion is low, medium or high.

For the cross validation releases of each project (that is, all the releases that are not the test release), we then come up with intervals for each project which will be used for the classification task.

Tables 11 and 12 show the quartile for all the release ids defined in tables 7, 8, 9, 10.

Based on the second and forth quantiles we convention on the intervals 9, 18 and infinite. We decided to take the intervals so that the ranges would be more equally distributed. While we could consider testing multiple values towards getting the best classification accuracy, we believe that in practice it would be more significant to the stakeholders of the prediction system what intervals are actually meaningful. Having a classifier that would get good accuracies on intervals that are meaningless wouldn’t be useful. In that sense, we opted for sacrificing accuracy for our legibility based on intuition, in the hope to observe how well the classifier would perform. Lastly, we chose the upper bound to be infinite because the highest discussion value could vary across releases. We can think of the interval that contains infinite, since this is a 3 scale point, as just "high" amount of discussion.

Under the same rationale, we chose the intervals cut points to be 0, 6 and Infinite for Pdfbox.

Having turning our discussion estimator to intervals, we are now able to apply machine learning classification algorithms using our file metrics and maintenance effort estimators to predict discussion.

5.4.1 Support Vector Machines

For Support Vector Machines we used a one versus all implementa-
Quartiles | Rel.ID 1 | Rel.ID 2 | Rel.ID 3 | Rel.ID 4 | Rel.ID 5 | Rel.ID 6
--- | --- | --- | --- | --- | --- | ---
0%  | 1  | 1  | 3  | 0  | 0.0 | 0
25% | 4  | 4  | 9  | 0  | 0.0 | 3
50% | 4  | 6  | 12 | 14 | 4.0 | 9
75% | 10 | 15 | 48 | 18 | 5.5 | 16
100%| 23 | 46 | 48 | 35 | 40.0| 50

Table 11: Derby Quartiles

Quartiles | Rel.ID 1 | Rel.ID 2 | Rel.ID 3 | Rel.ID 4
--- | --- | --- | --- | ---
0%  | 0.0 | 0  | 0  | 1
25% | 2.0 | 1  | 1  | 1
50% | 3.0 | 3  | 3  | 2
75% | 4.5 | 4  | 6  | 4
100%| 12.0| 19 | 6  | 11

Table 12: Pdfbox Quartiles

Predicted/Ground Truth | Low | Medium | High
--- | --- | --- | ---
Low | 58 | 9  | 0
Medium | 3  | 5  | 3
High  | 1  | 2  | 3

Table 13: Derby Support Vector Machine Confusion Matrix

3 We use a one versus all implementation because support vector machines classify only binary labels. Since we have three class labels (low, medium and high), for each original data table input to the support vector machine, we create three tables such that now the classes only contain zero or ones values. For the first table, low has its value set as ones, while the remaining are set as zeros. For the second table we set it as one and the remaining class as zeros, and we do the same for high. In a sense, we are creating three support vector machines for each data input to our original support vector machine. Each one of three support vector machines learns how to classify a distinct class label accordingly. This way we extend a binary classification support vector machine to a multi class by means of using three or more support vector machines.

<table>
<thead>
<tr>
<th>Predicted/Ground Truth</th>
<th>Low</th>
<th>Medium</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Medium</td>
<td>7</td>
<td>27</td>
<td>1</td>
</tr>
<tr>
<td>High</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 14: Pdfbox Support Vector Machine Confusion Matrix

<table>
<thead>
<tr>
<th>Predicted/Ground Truth</th>
<th>Low</th>
<th>Medium</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>12</td>
<td>9</td>
<td>46</td>
</tr>
<tr>
<td>Medium</td>
<td>0</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>High</td>
<td>0</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 15: Derby Feed Forward Back Propagation Artificial Neural Network Confusion Matrix

<table>
<thead>
<tr>
<th>Predicted/Ground Truth</th>
<th>Low</th>
<th>Medium</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Medium</td>
<td>0</td>
<td>32</td>
<td>3</td>
</tr>
<tr>
<td>High</td>
<td>0</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 16: Pdfbox Feed Forward Back Propagation Artificial Neural Network Confusion Matrix

<table>
<thead>
<tr>
<th>Predicted/Ground Truth</th>
<th>Low</th>
<th>Medium</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>47</td>
<td>5</td>
<td>19</td>
</tr>
<tr>
<td>Medium</td>
<td>1</td>
<td>1</td>
<td>17</td>
</tr>
<tr>
<td>High</td>
<td>0</td>
<td>1</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 17: Derby Random Forests Confusion Matrix

5.4.2 Artificial Neural Networks

5.4.3 Ensemble Methods

5.4.3.1 Random Forests

5.4.3.2 AdaBoost M2

5.4.3.3 Bagging

5.4.4 Non-interval prediction

For the sake of "completeness", we asked ourselves if running the models without defining intervals, that is, attempting to estimate the pre-
<table>
<thead>
<tr>
<th>Predicted/Ground Truth</th>
<th>Low</th>
<th>Medium</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Medium</td>
<td>0</td>
<td>33</td>
<td>2</td>
</tr>
<tr>
<td>High</td>
<td>0</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 18: Pdfbox Random Forests Confusion Matrix.

<table>
<thead>
<tr>
<th>Predicted/Ground Truth</th>
<th>Low</th>
<th>Medium</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>33</td>
<td>2</td>
<td>32</td>
</tr>
<tr>
<td>Medium</td>
<td>0</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>High</td>
<td>0</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 19: Derby Adaboost M2 Confusion Matrix.

<table>
<thead>
<tr>
<th>Predicted/Ground Truth</th>
<th>Low</th>
<th>Medium</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Medium</td>
<td>0</td>
<td>33</td>
<td>2</td>
</tr>
<tr>
<td>High</td>
<td>0</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 20: Pdfbox AdaBoost M2 Confusion Matrix.

<table>
<thead>
<tr>
<th>Predicted/Ground Truth</th>
<th>Low</th>
<th>Medium</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>41</td>
<td>3</td>
<td>23</td>
</tr>
<tr>
<td>Medium</td>
<td>0</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>High</td>
<td>0</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 21: Derby Bagging Confusion Matrix.

<table>
<thead>
<tr>
<th>Predicted/Ground Truth</th>
<th>Low</th>
<th>Medium</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Medium</td>
<td>0</td>
<td>33</td>
<td>2</td>
</tr>
<tr>
<td>High</td>
<td>0</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 22: Pdfbox Bagging Confusion Matrix.
cise measure, would give us significant results. While we believe the results would be less meaningful to the stakeholders, this would decrease the threat of our interval definitions affecting the precision of our results. We also asked ourselves if using purely structural complexity metrics, instead of considering two of the effort estimators as predictions, would give significant results.

Another motivation for this is that in creating intervals, we were uncertain if the impossibility of prediction of high and low intervals of discussion were due to the low amount of data points, or a particular characteristic of those intervals.

In considering the exact values for prediction, we can not use confusion matrix for the method evaluation, but instead an error function. We used the *Mean Squared Error* function, since the implementation of the used learning algorithms use this method to train.

To reason about the error function, we must ask ourselves how off the expected valuable it is tolerable to accept the prediction given the error method.

We have therefore created three neural network models, choosing as training and evaluation models the previous choices of releases. All three models contain as predictors the structural complexity metrics, but differ on what they attempt to predict. Each model predicts either *churn*, *actions* or *discussion*.

We observed that using *structural complexity metrics* as predictors, the model was unable to learn how to predict *churn* (the error exceeded at least $10^4$ lines of code for both Pdfbox and Derby projects). However, the other two models were able to learn to predict discussion and actions. Specifically, the Derby models were able to predict the amount of actions and discussion with an error on average below 4 (comments or amount of actions), while the pdfbox project models were able to predict with an error below 1 (comments or amount of actions).

In considering the error interpretations, and our understanding of the data distribution, we believe that Derby amount of actions is not acceptable, given that on average the amount of actions are usually 1 or 2. However, for the remaining predictions we believe the error to be acceptable, even for discussion on 4 comments (given the previous shown histograms in this chapter).

### 5.4.5 Other models

We have also attempted to use statistical methods such as poisson regression and its variants (negative binomial distribution and zero inflated models), in collaboration with the first author of [20], however we were unable to get a good error performance on any of the models. The later model was particular of our interest, given that it is able to predict in a single model all the three effort estimators in the
same time. According to the results, churn high variability in comparison to the structural complexity file metrics were the potential cause of such errors. Since splitting the model was feasible using machine learning methods, we decided to not push this analysis path forward.

A last attempt was made in predicting churn, in considering the variability issue using learning methods by normalizing the values. This made it possible to predict churn on both projects with an acceptable error of less than one unit. The problem on this approach is that we are not only interested in prediction with lower error rates, but also be able to understand the values. Therefore, while it is possible to learn from the pattern in normalized input, the model is not viable due to the inviability of the results interpretation.

5.5 CHAPTER SUMMARY

In this chapter we presented different prediction analysis using our obtained data. The histogram analysis gave us insight over the distribution of the core category effort estimator discussion. Correlation analysis enabled us to better understand the relationships of our variables, and the prediction analysis answered the question of whether or not we are able to predict indirectly effort from structural complexity metrics.

We concluded that we are able to partially predict effort from structural complexity metrics, being churn the only effort estimator whose error across both projects are unacceptable due to its high variability or normalization. Given that churn also affected the feasibility of prediction of all three proxy effort estimators, this may be an indication that churn sets apart as an effort estimator from discussion and actions in respect to structural file complexity, or that the pattern differs from churn to discussion and actions. Since the kinds of patterns that can be identified are associated to the statistical or machine learning being used, further investigation are needed to confirm these hypothesis, which are outside the scope of this work.

On the next chapter we discuss the general conclusions of the framework, its limitations and future work.
Part IV

CONCLUSION AND FUTURE WORK
CONCLUDING REMARKS

From the moment I fell down that rabbit hole
I’ve been told where I must go and who I must be.
I’ve been shrunk, stretched, scratched, and stuffed
into a teapot. I’ve been accused of being Alice and
of not being Alice but this is *my* dream.
“*I’ll* decide where it goes from here.

— Alice - Aline in Wonderland - Lewis Carroll

6.1 CONCLUSIONS

In this work, we have proposed and shown the feasibility of applying a flexible framework that is able to extract effort estimators from a given domain, quantify them by means of a mined dataset, and using learning and statistical algorithms to conduct prediction.

By applying the framework to OSS, we have observed that aside from it is main goal, which is predict effort, we have obtained and contributed to current state of the art work in a number of ways: We proposed an alternative framework that is not alienated to the domain and yet does not use anecdotes to choose variables; we identified an extensive number of threats to validity across the framework pipeline, whose interactions and implications to the next or previous phases were not previously identified altogether; we were able to predict the core category effort estimators.

Contrary to most of current work in software engineering in mining repositories, we have made available all the reports and logged data to assess validity. Currently, this is not an adopted practice as cited previous in our literature work, although it is considered important or required in other applied fields of statistics and machine learning.

6.2 LIMITATIONS

As with any research result, the results are not conclusive, and further investigation of applying this framework in other domains are necessary. Furthermore, the framework has a number of limitations that are important to be highlighted.

First, grounded theory is an extremely laborious and error prone method, in comparison to other automated analysis methods that have been applied in this work. In practice, a workable theory is expected to require few months before any sense can be made out of the analysis. Intermediate results are also not recommended by the
method. This is a price that is currently paid for moving away from a pure anecdotal approach.

Second, there are numerous threats in acquiring datasets that are usually neglected, as previously stated in chapters 3 and 4. The viability of the framework is therefore limited to the availability of the data, and the quality of the data obtained. It is also important to note, as it is pointed out on the preceding work to this [75], that mining project datasets also require large quantities of manual work. Specifically, project choice, schema organization, data access quotas, long waiting times to identify code bugs (since it requires a full mining step to be completed), among others makes acquiring a single project an incredible challenging task.

Third, the predictions models employed in this work are not exhaustive. In contrast to what is suggested in our findings of our literature review (chapter 3), we did not employ or verified the best ensemble of methods currently stated in state-of-the-art. In practice, evaluating each method requires in-depth knowledge of both data and method, which is seldom the case in literature on the data part, due to the lack of qualitatively analysis of the log as pointed out in the data concerns section of 3.

Forth, in considering practical aspects, there is also a limitation of available algorithms in both open source tools and industry tools. In despite of using both R and MATLAB, well known tools for both statistical and machine learning analysis, we have found that not all algorithms were available, and evaluating each algorithm given the data understanding would require a separate work on its own right.

6.3 Future Work

We are considering extending the threats to validity section combining both qualitative and quantitative analysis, and analyzing the impact of different data configurations and their impact over the analysis. We believe this to be an important collaboration to current research practice in mining software repositories, given that best model choice for a given dataset is usually based on error measures as discussed in 3. Understanding the meaning of the error and it’s impact could provide further insight on improving learning algorithms for software engineering works.

We have also obtained industry data very recently to apply the framework. This particular dataset have a unique characteristic in that it guarantees all issue to file mapping exist (this occurs due to the company process). By removing this threat to our framework, we will be able to better assess its feasibility in other domains.

While accomplishing prediction of effort we have not experimented an important stage of research that is often missed, which is apply the results in practice in the environment where the research was moti-
vated. Having this work being motivated from OSS, we find it hard to apply this specific study case and observe if it improves the quality of the OSS projects. However, having being successful in predicting indirect effort measures, we are expecting to apply the framework to our industry partners in order to evaluate such important step of validation of the results.

Lastly, while statistical significance and error predictions are of importance, they are of little use if the stakeholders of the research goal are unable to make sense of it. An important phase that we will investigate in our framework is how to make meaningful visual graphics and dashboards or rules that can be used to improve the domain process while offering little impact over the project process.


[10] A. Bachmann and A. Bernstein. When process data quality affects the number of bugs: Correlations in software engineering datasets. In Mining Software Repositories (MSR), 2010 7th IEEE


[62] Kazuki Nishizono, Shuji Morisaki, Rodrigo Vivanco, and Ken ichi Matsumoto. Source code comprehension strategies and metrics to predict comprehension effort in software maintenance


This work was the result of endless online and offline discussions among many people I had the chance to collaborate with, all of which I am extremely thankful. It started as an informal remote collaboration between myself and the Hawaii and Drexel University group, and grew to include other universities and industry companies.

Final Version as of September 2, 2013 (classicthesis September).
DECLARATION

I hereby confirm that the present work is solely my own work and that if any text passages or diagrams from books, papers, the Web or other sources have been copied or in any other way used, all references - including those found in electronic media - have been acknowledged and fully cited.

Brazil, Bahia, Salvador, 2013

Carlos Vinicius Andrade
Silva
Part V

APPENDIX
This chapter includes a list of all theoretical memos of the grounded theory chapter. Memos generated directly from the data, memos generated out of memos, and memos generated after memo sort are described on the following sections. It is important to note that the order and sections they are presented here are not the order of which they were generated. These memos categories were extracted from the annotations of M, M*, M** made during the coding process for later tracking.

### A.1 Memos generated from data

This section describes the memos generated directly from data. They were identified from the annotations by the convention used of the prefix "M".

- **LUCENE-4_M:** Solution disagreement is followed by solution agreement. This seems to suggest motivation from the community to try to fix the bug instead of just denying answers and expecting solutions.

- **LUCENE-5_M:** This is a different model of solution. The initial user did not accuse a solution or seem eager to solve it, he/she just show the steps. Later, he (Gerhard) sought the solution and in lack of replies he fixed on his own. It starts to seem that the core development once in a while comes around to gather a final decision and then do something. When ‘desire for a final decision’ occurs apparently so far it can either go as ‘wont fix’ and as his own fix solution. So far it seems to depend on the inactivity of whoever initiated the bug.

- **LUCENE-DEV-27M:** Using discussions to turn it in documentation

- **LUCENE-9_M:** I believe this bug reporter calls attention that the logic theory behind it is wrong. It is a theoretical problem, not a code problem.

- **LUCENE-16-M:** I think there is a somewhat good amount of evidence in respect to how a context is provided so that other contributions can reproduce the problem, and they do vary. This one for instance is the first occurrence where I see little clue of what is the bug (although not the first that has no solution).
• LUCENE-7_M: There was no seek for agreement here. Perhaps it was too simple to look for one?

• LUCENE-44-M: So there is a source for users to play around with extensibility like a sandbox...

• LUCENE-43-M: The TODO list seems like it was a painkiller for people having issues being reported over and over and developers having to mark things as duplicate and link them over. Think of it, the statement of ‘and Lucene TODO contains a similar item’ might suggest that had the user check the feature list before posting the enhancement he wouldn’t (the developer) go to the hassle of checking this. Since this is just assumptions I can’t use this as code.

• LUCENE-37-M: At this point I am still uncertain on what motived the reporter to post bugs, they just appear with the solution. I think the improvements one have an air of despire, but not really why it annoys them.

• LUCENE-7_M: It depends on self awareness to notice something is being doing wrong or from someone else on the community. Here a patch is applied is applied with a different description of the bug and to solve a documentation bug(!). However the original author noticed he was doing it wrong.

• LUCENE-19-M: This bug is a variant of many of the others from which I recall. The other codes are bugs of which were more reproducible. This one required a specific situation of which it seems easier to contact directly the reporter to provide a final answer to this case. The issue thus depend solely on the reporter to know if it was indeed solved or not. One could consider this I kind of potential debt associated to the reporter status.

• Lucene-11-M: This is not the first time I see a solution being attached by someone else to the wrong issue. I need to find where I coded this before Issue 11. It seems that some people are always aware of those mistakes. Maybe they try before running, I don’t know.

• LUCENE-25-M: This suggests how they are able to trace the solution back to the problem in the code. This might identify and explain much of the start of the solution process.

• LUCENE-25-M: This is important, it is not of the strategies employed by the decision seekers to decide over closing or not the issue. This is not always the case, there are codes from where they just go on with their beliefs. This uncertainty in this case, seems to derive from the bugzilla migration as the answer is
replicated in JIRA-code. However, these are just assumptions which at this point are not grounded in data.

- LUCENE-20-M: Versions were not labeled formally when migrated from bugzilla. They are in the text, although in the affected versions they are not mapped.

- LUCENE-95-M: This appear to be a patch compatibility problem which generated a lot of discussion.

- M: An example of the necessary input is given here.

- LUCENE-41-M: Another evidence of how they see more specific information of the errors to track them down. This has occurred before.

- LUCENE-16-M: At this point I have gotten in mind already an workflow of how things proceed: An issue is brought, over different contexts and conditions by a DECISION SEEKER. Other DECISION SEEKERS, which having varying patterns of the defined PROPERTIES starts appearing to solve the issue. Agreement is seek, over voting, and an exchange of solutions. The most voted among the decision seekers are then attached to jira. Finally, someone with authorization applies it. Perhaps when someone says FIXED then it is a mark that the flow has ended.

- LUCENE-60-M: They were using the issue tracker for partnership.

- LUCENE-21-M: These kind of issues are of a different kind as well in the sense the description and title are totally misleading. The real problem (user ignorance), lies in the commentaries, not in its original description as well.

- Lucene-11-M: It doesn’t seems like this is something an average joe would be able to do. They require a unit test to address the problem from the repoter, its not like can someone state just that something is wrong, but require more than that. Like..the reporter must take responsibility on the work for solving the problem.

- Lucene-11-M: It seems that the involved people recognize when they should or not account for the underlying problem depending on if they have or not responsibility for it. Maybe there is something they rely on to make such distinctions? The reasoning is not evident anywhere.

- LUCENE-14-M: Still on the mysterious question on how one can tell how it has been fixed in the end. All that seems to be known is either that it has been fixed, and sometimes the person
is identified. I wonder if by fixed they mean someone applied 
the patch that was attached? So far the only guarantee is that 
every now and then they it has been fixed by saying that, and 
sometimes by whom.

- LUCENE-17-M: This is an interesting quote. It seems all those 
  concerns are mapped to Bugzilla days (given the Bugzilla label 
id). Here they disapprove improvement suggestions!

- LUCENE-3_M: Apparently this refers to LUCENE-2. Curiously 
  enough, Lucene-3 came after Lucene-2, instead Lucene-2 is con-
  sidered the duplicate of 3. The messages are the same, which 
suggest this duplication is of the type copy and paste. Since the 
issues from bugzilla differ, most likely the order of creation on 
Bugzilla is not enforced when moving to Jira (at least on the 
issue codes). No evidence has been found at this point.

- LUCENE-5_M: No solution was proposed this time, but a way 
  to reproduce and the output message

- LUCENE-3_M: This is one evidence that sub-tasking is logged 
  while performing other activities. I believe I could consider this 
  a refactor while fixing a bug later on if I find other kinds of 
evidence.

- LUCENE-12-M: This is interesting. The person reports that a 
  new bug has been opened and regret not reopening this one 
  for documentation. Traceability of the bug might be very hard 
because of this, however I need to see if more codes about this 
comes up.

- LUCENE-1_M: This is a linkage to bugzilla original issue, how-
  ever this code is invalid. Probably was migrated to JIRA.

- LUCENE-6_M: There is no reference to when the issue was 
  fixed, only belief. The issue might as well never been fixed al-
  though it was marked as so. Furthermore no traceability to the 
original solution.

- LUCENE-6_M: Sometimes some hidden codes appear from the 
  not read-by-line enforcement. For instance after I saw this part I 
thought "Hmm..this is a documentation problem", then I asked 
myself "how did a documentation problem was noticed?" When 
I looked back for the Description section line by line I noticed 
the passage "just as it says it does", which originally for me was 
simple a seek of agreement. Not only a seek of the agreement is 
being wanted (from the word ‘right?’) but as well the question 
embodies that documentation is being used as guidance!
• LUCENE-2-M: The reference of the comment refers to Bugzilla. Most likely was moved together. Reference for the Bugzilla issue is gone. Considering that this is still LUCENE-2, and LUCENE-1 is clearly not 4105, this might suggest this JIRA did not migrate all the issues.

• LUCENE-27-M: This seems to suggest that I should see how contributors aid users concepts in the mailing list.

• LUCENE-20-M: There are quite a good amount of duplicate of this issue. Perhaps this suggest popular bugs and not much concern from others to observe they are duplicated?

• LUCENE-78-M: This is an interesting use for unit tests, to make clients lose arguments to state a bug instead of just ensuring nothing has been broken.

• LUCENE-44-M: Pier Fumagalli set up this Apache Bug Database, perhaps there was an underlying concern for this, although the contributors doesn’t seem to like this.

• LUCENE-38-M: This looks like more evidence to say how technical debt is incurred in open source. A year old bug, due to "I don’t recall being mentioned on the list " is marked as won’t fix.

• LUCENE-9-M: This bug seems to suggest more conformance or lack of documentation problems and what it causes. There is no evidence on this text that there was lack of conformance of documentation, so this is only speculation.

• LUCENE-4-M: Notice the day between this comment and the previous comment solution! There is a time frame of over 5 months!

• Lucene-11-M: There has been a time gap of nearly one month since the comment request. It seems they are pretty patient on waiting for answers.

• LUCENE-13-M: Woah a 3 year to Solve bug, but the latest update is very recent!

• LUCENE-14-M: Contract is a new word around. I have been trying to understand how one could infer what would be expected or not from the system. This ‘contract’ seems to be the answer to everything. Documentation for sure is one and expected but I didn’t know about this. Perhaps I could say that JUDGEMENT from DECISION SEEKERS are justified (theoretical code) by UNKNOWN CONCEPT (the one that relate to documentation conformance) and CONTRACT?
• LUCENE-21-M: This is the second bug I see in respect to logic. One could understand this as a type of error that is different in the sense it is not the software, the documentation or anything else. It is the ignorance of the user on a knowledge that would expect from it (in this case Boolean logic).

### A.2 Memos of Memos

This section describes the memos generated out of memos. They were usually generated from the understanding of the data as being continuously processed from different sources of data, indicators and memos generated from data.

• M*: The assigner reported the bug AND suggested the solution

• M*: I haven’t see Otis or anyone from apache reporting bugs. Does this suggest they do not ever log the issues they fixed? This might have something to do when they say ‘fixed’ and do not attach patches to Lucene as well. They acknowledge for tracking purposes but the solution is not documented.

• M*: There is some pattern in respect to how bugs are reported and improvements. Among bugs there are people reporting bugs, those complaining because of their company, those who are eager to fix, those who report only the trace and so on. Improvement seems to have a different kind of contribution. It is like the contributor is caring more about enhancing lucene for ‘charity’ rather than being annoyed by a bug

• M*: Request for more feedback from the community (”Any words on this?”)

• M*: This has to do with the suggested solution in comments

• M*: This code doesn't seem right

• M*: Code for reproducing the error

• M*: Example of the necessary input

• M*: These are just informal simple tests

• M*: Demo code is given here

• M*: Solution was attached as a patch but not mention on description or comments for discussion

• M*: Seeing the other attributes and the category itself, the question ‘Can’t I tell from the data who are the seekers?

• M*: This is a judgement belief of fix based on other solutions.
• M*: Assigners belief of fix are suggestions of what they think would fix of the bug, based on logical judgement.

• M*: Iterative Informed as of the process of generating information iteratively is what wards of the blockers. In a sense, one may think that although few can mark an issue as won’t fix, the community have power to avoid such.

• M*: I have seen contributions and offer of contributions waiting on a reply for deciding for a contribution. Perhaps the awareness and proactiviness of those responsible for lucene is a major role on deciding the projects life. I will probably only find those on the late days of a dying project.

• M*: There is a problem with this inference. The update is very recent but I don’t know why. Might be due to the Jira Migration. I will ignore this evidence.

• M*: The evolutionary process relies on the user community. It goes by chance that others collaborate reporting bugs, making enchantments, suggesting features and so on. I think one could say that they put the project in the right path on what is needed by the majority of the users and what is mostly affecting its usage. I would need to compare against a DYING project to learn more about this.

• M*: There is another idea that is important to convey here is that it is by chance that small contributions are taken. It is not like some major task is defined and then a lot of people perform. Instead someone offers a little bit of contribution which is incorporated. Another offers another small bit that is incorporated. And it is from small and small bits that when so many are offered make the project prospect. Once in a while someone makers a more bigger offer like an enhancement. This sort of reminds me of a church or anything in the society that lives out from donations (What do others do to receive more donations?)

• M*: Maybe it is the case that the small evolutions by chance can only occur within the timeframe the user send the email. For instance, on the motivation of the person requiring attention around lucene-dev-40s there is a urgency before the release is performed. In that sense, the ‘chance’ of the core theory might be able to associate to a time frame. Small Evolution Within Time Frames would be then a better suited name rather than ‘by chance’. or maybe both...since I need to incur the meaning of things happening due to unknown influences of each individual who comes to Lucene.
• M*: Up to issue 71 in lucene I have been getting the impression that the pattern is client versus collaborators interfacing in JIRA (which actually comes from Bugzilla moved issues).

• M*: 1 Year old bugs seems to be born of Decision Seeker, when there is no motivation to seek a final decision.

• M*: Strange update date, will ignore this one for now

• M*: This doesnt really mean anything for old issue.

• M*: Ignored as well. Not their comment that it was old, I should ignore my self impressions.

• M*: Ignored as well since it is on my assumption

• M*: Doesnt seem relevant.

• M*: The judgements of decision varies based on the participant impression. They vary from an accurate decision with a reason being stated to belief based on their impressions. Impressions are based on patch analysis and the discussion. Need to rework on code to see more details.

• M*: It seems the deny of fixing a bug occur as a concept, need to investigate this.

A.3 SORTED MEMOS

The order of the memos in this section matter. They indicate the order of which the memos were sorted to be then further presented in writing, as previously show in the grounded theory chapter. Memos in this section were generated as of reviewing the memos of previous section and organized substantive codes of the trees.

• M** The concern seems to be Problem Solving according to LUCENE-4_M and LUCENE-5_M.

• M** What is IIC?

• M** IIC motivates consent solution by majority, but improvement over previous solution based on continuous state of facts. Bad solutions are discarded and understood as to seek of optimal choice.

• M** A chance of need is necessary for project survival. This chance of need is a manifestation of external problems particular to people that reflect in the appearance of concerns to be solved during problem solving. It can range from a company deadline (see M* on company deadline) to simple personal need.
• M** Some dimension affects the seek for agreement. What it is? (LUCENE-7_M).

• M** Solution Agreement and Disagreement covariate during problem solving (LUCENE-4_M).

• M** There are different degrees of motivation of which the participants collaborate with each other (LUCENE-5_M).

• M** Problems may be logical or code (LUCENE-9_M).

• M** Request and Will of opinion and understanding seeking the optimal solution occurs.

• M** While no final solution occurs, a self verification and IIC over uncertainty occurs in order to reach a Clarity Conclusion.

• M** Clarity of Conclusion is required to assess when IIC is over (LUCENE-37-M),(LUCENE-19-M),(LUCENE-11-M). They manifest themselves on different types as certainty that the problem has been completed, evaluation that the problem has not be answered with the wrong solution, dependency on other party for conclusion, duplicated issue identification(LUCENE-20-M).

• M** Uncertainty of Conclusion

• M** IIC occurs in varying forms. Partnership (Lucene-60-M), Problem Solving, Trading, or they might not occur. Lack of occurrence are a manifestation of simple problems or authority.

• M** IIC is a balance between agreements and consents. Time for response and proactiviness affects the flow of discussion and furthermore the solving concern. Considerable delay in the continuation of IIC initially leaves to problem death in a small scope, and later to project death.

• M** Iteration Blockers are based on Activiness, Lack of time, and Effort. Solving of the concern may also occur without IIC by deny of report, refusal of responsability, impossibility of replication, continual discussion, or expectation and understanding of conformance.