Comparison of IDE extracted effort versus static metrics for assessing software maintainability

Michel Kinson
kinson.michel@student.uva.nl

August 22, 2017, 40 pages

Supervisor: Dr. Aiko Yamashita and Prof.dr. Jurgen Vinju
Host organisation: CWI: Software Analysis and Transformation (SWAT)

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

Contents

Abstract

Software cost estimation is crucial in software project lifecycles. Inaccurate estimations can lead to devastating results. Previous studies have used software metrics as proxies for maintenance effort (i.e. time spent on a unit of work). Previous studies have the assumptions that large classes and more complex classes require more maintenance effort. However, the relationship between these proxies and actual maintenance effort has not yet been empirically examined in detail. The main goal of this thesis is to assess whether it is possible to accurately predict maintenance effort using total Source Lines of Code (SLOC) and Cyclomatic Complexity (CC) of classes.

To accomplish this goal, we reused a dataset acquired from a different study on the impact of code smell over maintenance effort. In the aforementioned study, six professional developers were hired to perform three maintenance tasks on four functionally equivalent Java Systems. Each developer performed three maintenance tasks. While working on the tasks, IDE activity logs were collected. In this study, we created a log analysis tool (LAT) to extract effort from developers' activity logs and investigate the effects of SLOC and CC on maintenance effort. We created and empirically validated LAT for accuracy using the dataset from the previous study. The results show that neither total SLOC nor CC are sufficient to predict maintenance effort. Furthermore, using SLOC by itself or together with CC result in the same adjusted R^2 of 0.21.

One major contribution of this thesis is that it demonstrates that activity logs are reliable data sources for measuring software maintenance effort. That could lead to more focus on collecting and analyzing activity logs to support better estimations on maintenance effort.

Chapter 1

Introduction

1.1 Motivation

There is never enough time or money to cover all the desired features we would like to put into our software products. To prioritize the features, we need to understand the costs (effort) behind each of them, as well as their criticality from a business point of view. Thus, software cost estimation is a crucial task in the software project life cycle. Bakır et al. [5] defined software cost estimation as the process of predicting the effort required to develop a software project. By effort, we mean the amount of time spent to finish a unit of work. Such estimation can help project managers in planning and resources allocation. Inaccurate estimations can lead to devastating results. For example, over estimation can lead to waste of resources, rejecting other projects, thus threatening organization competitiveness. Conversely, underestimations can lead to a lack of resources allocation, schedule, and budget overruns, which can ultimately lead to project failure.

Because effort estimation is so crucial, it is a very active area of research. Jørgensen [31] explains the different types of methods for estimating effort:

- Expert-based methods that uses human expertise, possibly augmented with process guidelines, checklists and data (e.g. documented data from previous unit of work) to generate predictions.
- Model-based methods that can summarize old data via data miners to make predictions over new projects.
- A hybrid method that combine expert and model based methods.

Effort estimation normally is done at a task level, because it is difficult to record manually effort at file/class/method levels. However, software metrics can support estimation of maintenance effort at these levels. Examples of such metrics are Source Lines of Code (SLOC) and Cyclomatic Complexity (CC). Developers use these metrics to gain some insight into their code quality as proposed by Zuse [63] and Fenton and Pfleeger [18]. For example, the larger the code, the more time is used to understand and to perform changes. In the same way, the more complex a piece of code is, the more it takes to understand it, thus to make modifications on it. For example, if there is a significant difference between the number of SLOC and CC of two classes, then we could assume that maintenance effort of the class with the highest SLOC and CC will be higher. Visser et al. [58] and Rosenberg et al. [47] echoed the same assumptions about SLOC and CC respectively. Visser et al. [58] and McCabe [39] explained that higher levels of CC affect maintainability negatively. Rosenberg et al. [47] noted that the higher the total CC of a class, also known as Weighted Methods per Class (WMC), the less maintainable the class would be.

So what is maintainability? Maintainability is formally defined by ISO/IEC [28, p. 10] as: *The capability of the software product to be modified. Modifications can be corrections, improvements or adaptation of the software to changes in an environment, and in requirements and functional specifications.* The ISO standard did not define a solid model on how to measure maintainability, thus many models have been suggested [26, 45, 10]. Code smells [1], SLOC and CC examples of source code attributes that have been proposed to measure software maintainability according to these models. These attributes are measured via static analysis tools. Examples of static analysis tools are cloc[2] and Sonar Qube[3]. One known drawback of source code attributes is that they constitute proxies for maintenance effort, but they do not constitute an empirical measure, such as effort (time). Also, they do not take developers' experience and familiarity with the system and language into account.

In expert-based prediction, static analysis tools can be used to assist in decision making. Static analysis is the process of analyzing code without executing it. Emanuelsson and Nilsson [16] defined it as an automatic method used to reason about runtime properties of program code without actually running it. Emanuelsson and Nilsson [16] also shows many tools that do static analysis. Heitlager et al. [26], Oman and Hagemeister [45] and Coleman et al. [10] are examples where static analysis is used to predict software effort. A major challenge of metrics-based assessment of source code is that empirical data on the effort (time) used to perform different activities (e.g. search, read, write, refactor) on a given project is often not available, as explained by Sjøberg et al. [52] and Soh et al. [55]. Consequently, it has been hard to relate maintenance costs empirically to measurable characteristics of software.

Measuring effort manually at file level is not feasible from a practical perspective because it would be tedious and prone to human errors. Therefore analysis of activity logs is a potential approach, since it can measure the actual effort that was spent on a file. This, in turn can be use to estimate effort, instead of using proxies such as SLOC and CC. However, there are potential issues to measuring effort from logs according to previous work. Sjøberg et al. [52] noted that it has been historically difficult to verify if log-based effort extractions are accurate because of the lack of grounded truth or empirical measurements of effort during a sufficiently extended period. Deligiannis et al. [13]'s study is an example, where the observational period only lasted for one hour and a half.

Sjøberg et al. [52] studied the impact of code smell over maintenance effort. In this study, six professional developers were hired to perform three maintenance tasks on four functionally equivalent Java Systems. Each developer performed three maintenance tasks. While completing the tasks, metadata was collected. Videos were taken during think aloud sessions and IDE activity logs were collected. The think aloud sessions were used to annotate the IDE activity logs with the goal of validating the annotation. By think aloud session we mean when researchers ask participants to think out loud while performing a task[17].

In this thesis, we revisited Sjøberg et al. [52] study since it counts with both: grounded truth data and activity logs where the observational period lasted three to four weeks. This enabled us to test if is feasible in practice to calculate effort by analyzing IDE activity logs.

1.2 Research questions

This thesis will attempt to assess:

1. How accurate can file-level effort measurements be when based on IDE activity logs analysis?

2. Can we effectively use previously measured SLOC and CC of a class to estimate/predict future maintenance effort on source code at class level?

[1]Fowler and Beck [22] noted that code smells embody poor design choices.
[2]e.g., http://cloc.sourceforge.net/
[3]http://www.sonarqube.org/

1.3 Contributions

Contribution #1

A log analysis tool (LAT) could potentially help identifying high levels of maintenance effort needed to work with an artifact. It could also be used by tool vendors to identify problematic artifacts and provide real-time feedback/support to developers during software maintenance activities.

Contribution #2

If LAT can demonstrate that activity logs are reliable data sources for measuring effort, that could lead to more focus on collecting and analyzing activity logs to support better estimations on maintenance effort.

Contribution #3

If we can estimate maintenance effort at the class level (via the analysis of activity logs), then we could examine different attributes of a class to determine which of them lead to higher or lower effort.

1.4 Thesis outline

The thesis is organized as follows. Chapter 2 provides background information and provide related work on the thesis' topic. Chapter 3 presents the methodology that was employed to accomplish the goal of this thesis. The methodology chapter divided into two parts: 1) the implementation and validation of LAT, and 2) the empirical study used to answer the second research question. Discussions and results on the work conducted are presented in Chapter 4. Chapter 5 concludes the thesis and discusses avenues for future work.

Chapter 2

Background and related work

This chapter will present the background information and related work concerning this thesis. We clarify terminologies such as static analysis, software maintainability, software maintenance effort, software effort estimation, and event logs collection tools.

2.1 Static analysis

Wool has a tendency to collect static electricity and thus to attract dust and lint. Developers know that programs have a similar tendency to attract defects.
- Louridas [38]

Static analysis is the process of analyzing code without executing it. Emanuelsson and Nilsson [16] define it as an automatic method use to reason about runtime properties of program code without actually executing it. Static analysis can measure class properties such as code smells (i.e. God Class, God Method, Data Clump, and much more.), size and complexity. These measures are used as proxies for maintainability. SLOC and CC are two common metrics used by developers as proxies for maintenance effort. In listing 2.1, the code snippet has 7 LOC/SLOC and has a CC of 2.

Source Lines of Code (SLOC): Conte et al. [11, p. 35] defined SLOC as any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statement.

Cyclomatic Complexity (CC): McCabe [39] explained that CC of a program is the maximum number of linearly independent circuits in the control flow graph of said program, where each exit point is connected with an additional edge to the entry point.

Weighted Methods per Class (WMC): Rosenberg et al. [47] explained WMC metric as the sum of the complexities of all class methods. Basically $WMC = \sum CC$.

```
public double sqrt(int n) {
  // Newton-Raphson method
    double r = n / 2.0;
    while (abs(r - (n / r)) > 0.00001){
      r = 0.5 * (r + (n / r));
    }
    return r;
}
```

Listing 2.1: Example of SLOC and CC definition

Much has been written about the apparent linear correlation between CC and SLOC. However, in a recent large-scale study by Landman et al. [35], the evidence towards this conclusion has been refuted. Therefore, In the current thesis, we consider that these two metrics are sufficiently independent variables to warrant their independent investigation.

Many have suggested that the larger the class, the more time is used to understand and to perform changes. In the same way, the more complex a class is, the more it takes to understand it, thus to make modifications on it. If there is a significant difference between the number of SLOC and CC of two classes, then we could assume that maintenance effort of the class with the highest SLOC and CC will be higher. Visser et al. [58] and Rosenberg et al. [47] echoed the same assumptions about SLOC and CC respectively. Visser et al. [58] and McCabe [39] explained that higher CC affects negatively maintainability. Rosenberg et al. [47] noted that the higher the total CC of a class, also known as Weighted Methods per Class (WMC), the less maintainable the class would be.

2.2 Software maintenance effort

You can not control what you can not measure.
- DeMarco [14]

Cambridge Dictionary [8] defines "effort" as physical or mental activity needed to achieve something, or an attempt to do something. Within software engineering research, effort is defined as the amount of time spent to finish a unit of work [52, 41, 37]. So how is effort currently measured in software engineering studies?

Table 2.1: Effort measuring methods

Type	**Description**
Video recording [54, 52]	Video captures of developers' screens
IDE interaction data [54, 52]	Event logs of developers actions in the IDE
Manual recording [59]	manually keeping daily records

Table 2.1 illustrates various studies that have measured effort in their experiments [54, 52]. Wiegers [59] showed how it is done manually in the industry where records are kept of the time spent on a unit of work. Soh et al. [54], Sjøberg et al. [52] explained two methods (video recordings and IDE instrumentation) that have been used in software engineering experiments. Video software tools including VLC[1] and QuickTime[2] have been used during think aloud sessions. Ericsson and Simon [17] noted that think-aloud session is when researchers ask participants to think out loud while performing a task. In the following sections 2.4 and 2.5 respectively, we will describe typical effort estimation approaches and details on effort measurement by IDE instrumentation means.

2.3 Software maintainability

Who wrote this class?? I can not work like this!!
- Any programmer

Imagine having two softwares (software A and software B) with the same functionality. That means, given the same input, they both result in the same output. Software A's source code is easy to modify and require less effort given a new task. Software B's source code is barely understandable, let alone modifiable. Even though software A and software B have the same functionality, their quality differs. Software A maintainability is higher than the maintainability of Software B.

[1]https:www.videolan.orgvlcindex.html
[2]https:support.apple.comdownloadsquicktime

Maintainability is formally defined by ISO/IEC [28, p. 10] as: *The capability of the software product to be modified. Modifications can be corrections, improvements or adaptation of the software to changes in an environment, and in requirements and functional specifications.* The ISO standard did not define a solid model on how to measure maintainability thus many models have been suggested.

Kitchenham et al. [33] proposed four domain factors that influence the maintenance processes: product, organizational process, maintenance activities, and people. High emphasis has been given to the product and process factors in software engineering research. For example, in the context of estimating maintenance effort or maintainability assessments. Process-centered approaches for maintenance effort estimation can be found in the following references. [21, 24, 37]. Many of the process-centered approaches use process-related metrics or historical data to generate estimation models. Product-centered approaches for estimating maintenance effort or assessing maintainability include those discussed in Refs. [1, 6].

2.4 Software effort estimation

Software cost estimation is crucial in software project lifecycle. Bakır et al. [5] defined software cost estimation as the process of predicting the effort required to develop a software project. Such estimation can help project managers in planning and resources allocation. Inaccurate estimations can lead to devastating results. For example, over estimation can lead to waste of resources, rejecting other projects, thus threatening organization competitiveness. Conversely, underestimations can lead to a lack of resources allocation, schedule, and budget overruns, which can ultimately lead to project failure.

Effort estimation is a very active area of research because of its importance in software project lifecycle. Jørgensen [31] explains the different types of methods for estimating effort: Expert-based methods, Model-based methods and hybrid methods. Examples of Expert-based methods for effort estimation include the following best practices explained Jørgensen [31]: combine estimates from different experts and estimation strategies, assess the uncertainty of the estimate, ask the estimators to justify and criticize their estimates and use documented data from previous development tasks. Model-based methods range in complexity, from relatively simple *nearest neighbor methods* [32] to the more intricate *tree-learning methods*, as used in CART [7] to even more complex search-based methods that make use of tabu search to set the parameters of support vector regression [12]. Examples of hybrid methods for effort estimation include those presented in Refs. [9, 56, 51].

2.5 Event logs collection tools

Overall many stakeholders benefit from capturing and analyzing IDE activity logs. Snipes et al. [53] explained the following: First, IDE vendors leverage the data to get insight into ways to improve their product based on how developers use the IDE in practice. Second, researchers develop IDE activity logs collectors and conduct rigorous experiments to (1) make broader contributions to our understanding of developers coding practices and (2) improve the state-of-the-art programming tools (e.g. debuggers and refactoring tools). Finally, developers benefit from the analysis conducted on the IDE activity logs because these analyses lead to more effective IDEs that make developers more productive. Robillard et al. [46, p. 176] showed the following IDE instrumentations examples:

- Mylyn uses IDE activity logs to recommend source code artifacts relevant for a current task.
- OCompletion improves code completion tools based on a fine-grained analysis of previous edit interactions.

In this thesis, analysis of IDE activity logs produced by IDE instrumentation is a potential approach for measuring effort once developers' events are logged. As mentioned in the previous section, there are many ways to measure maintenance effort. Manual recording is one approach. However, measuring effort manually at file level is not feasible from a practical perspective because it would be tedious and prone to human errors. The effort extracted from the analysis of IDE activity logs can be a extremely beneficial for effort estimation. Instead of using proxies such as SLOC and CC, the actual effort can be used to build models that can predict future maintenance effort. However, there are potential issues to measuring effort from logs according to previous work. Sjøberg et al. [52] noted that it has been historically difficult to verify if log-based effort extractions are accurate because of the lack of grounded truth or empirical measurements of effort during a sufficiently extended period. Deligiannis et al. [13]'s study is an example, where the observational period only lasted for one hour and a half.

There exist a series of tools or software implementations that enable the collection of IDE activity logs. Table 2.2 shows a summary of data collection tools that were examined for the purposes of this thesis. In the remainder of this section, Mylyn and FLUORITE will be discussed in detail.

Table 2.2: Summary of Analysis on log collection tools.

#	Tool Name	Advantages	Disadvantages
1	Eclipse IDE activity logs Collector (UDC)	Well tested, widely deployed.	Collects only data on tools; sometimes missing data
2	Mylyn Monitor	Collects data both about tools and the program elements the tools are used on.	No details about code beyond element names collected. Extra work required
3	CodingSpectator	Very detailed information collected.	Information collected largely customized to observe the usage of refactoring, tools and it is not up to date. Legacy plugin
4	CodingTracker	Not available any more	No available any more
5	Fluorite	Already been used in previous projects, so it is stable. It captures fine grain of data such as time and selected artifact.	It creates one whole file per day. That file may become very large.
6	Mimec	Not available any more	Not available any more
7	WatchDog	Available for eclipse	Only provide high-level log data such as user active, reading, writing. No information on the specific file
9	Building your own	A high degree of customizability.	Extra work required to collect a wider variety of data

Mylyn Monitor

Murphy et al. [44] created Mylyn, a task-focused user interface, a top-level project of the Eclipse IDE that is part of many of the Eclipse IDE configurations. To better support developers in managing and working on multiple tasks, Mylyn makes tasks a first class entity, monitors a developer's interaction with the IDE for each task and logs it in a so-called task context. While the source code of the Mylyn Monitor can still be found on-line, it is not an active part of the Mylyn project anymore.

Data collected by Mylyn: Mylyn captures three types of IDE activity logs: the selection of elements, the editing of elements and commands in the IDE, such as saving or refactoring commands. These interaction events are monitored and then stored in XML format in a log file. See figure 2.1 for a log example of a developer selecting a Java class TaskEditorBloatMonitor.java in the package explorer of the Eclipse IDE.

```
<InteractionEvent
    StructureKind="java"
    StructureHandle="=org.eclipse.mylyn.tasks.ui/src&lt;org.eclipse.mylyn.
        internal.tasks.ui{TaskEditorBloatMonitor.java"
    StartDate="2012-04-10 02:05:53.451 CEST"
    OriginId="org.eclipse.jdt.ui.PackageExplorer"
    Navigation="null"
    Kind="selection"
    Interest="1.0"
    EndDate="2012-04-10 02:05:53.451 CEST"
    Delta="null"
/>
```

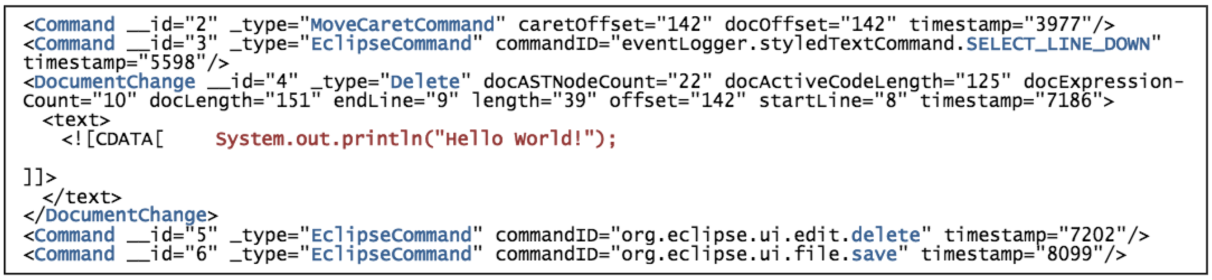
Figure 2.1: Example of a log generated by Mylyn monitor
[53]

Figure 2.1 demonstrates that Mylyn logs contain all the required information to be able to extract effort (time), as it has the selected artifact with a start timestamp and an end timestamp. Unfortunately, the code for the Mylyn Monitor is not part of the active Mylyn project anymore, although the code for the monitor and example code can be found in the incubator project online[3] [4].

FLUORITE

FLUORITE[5] is an event logging plug-in for Eclipse, which captures all the low-level events when developers use the Eclipse code editor. FLUORITE captures not only what types of events occurred in the code editor, but also more detailed information such as the inserted and deleted text and the specific parameters for each command [60].

Data collected by FLUORITE: FLUORITE captures three types of developer interactions with the Eclipse development environment: Commands, document changes and annotations. Figure 2.2 shows an example of a log snippet where the developer (1) moved the cursor by clicking mouse button, (2) selected one line by SHIFT + DOWN, (3) deleted selected code using the DELETE key, and (4) saved the file. Each event has its own parameters, and the whole deleted text is listed in DocumentChange event. Once FLUORITE[6] is installed on Eclipse, it begins to capture all the low-level events occurring in the code editor, and saves the transcript as an XML file when Eclipse is closing. FLUORITE is publicly available and still works in Eclipse.



```
<Command __id="2" _type="MoveCaretCommand" caretOffset="142" docOffset="142" timestamp="3977"/>
<Command __id="3" _type="EclipseCommand" commandID="eventLogger.styledTextCommand.SELECT_LINE_DOWN"
timestamp="5598"/>
<DocumentChange __id="4" _type="Delete" docASTNodeCount="22" docActiveCodeLength="125" docExpression-
Count="10" docLength="151" endLine="9" length="39" offset="142" startLine="8" timestamp="7186">
  <text>
    <![CDATA[     System.out.println("Hello World!");

]]>
  </text>
</DocumentChange>
<Command __id="5" _type="EclipseCommand" commandID="org.eclipse.ui.edit.delete" timestamp="7202"/>
<Command __id="6" _type="EclipseCommand" commandID="org.eclipse.ui.file.save" timestamp="8099"/>
```

Figure 2.2: Example log generated by FLUORITE.
[53]

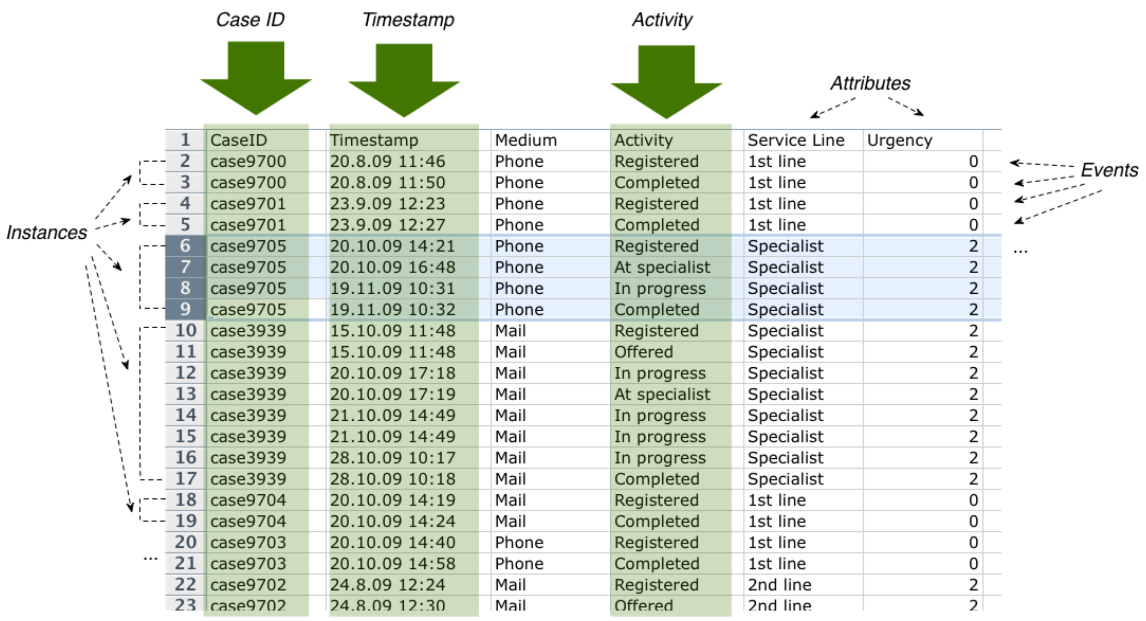
[3]http://git.eclipse.org/c/mylyn/org.eclipse.mylyn.incubator.git/tree/
[4]http://wiki.eclipse.org/Mylyn_Integrator_Reference#Monitor_API
[5]Full of Low-level User Operations Recorded In The Editor
[6]Fluorite can be found at https://github.com/yyoon/fluorite-eclipse

2.6 Process Mining

There are different approaches to extract effort from IDE activity logs. One approach is to subtract the end timestamp to the start timestamp. In this current thesis, a process mining approach will be used.

Van der Aalst and Weijters [57] defines process miming as follows:

Process mining is defined as the method of distilling a structured process[7] *description from a set of real executions.*



Case ID
Timestamp
Activity
Attributes
Events
Instances
...
...

1	CaseID	Timestamp	Medium	Activity	Service Line	Urgency
2	case9700	20.8.09 11:46	Phone	Registered	1st line	0
3	case9700	20.8.09 11:50	Phone	Completed	1st line	0
4	case9701	23.9.09 12:23	Phone	Registered	1st line	0
5	case9701	23.9.09 12:27	Phone	Completed	1st line	0
6	case9705	20.10.09 14:21	Phone	Registered	Specialist	2
7	case9705	20.10.09 16:48	Phone	At specialist	Specialist	2
8	case9705	19.11.09 10:31	Phone	In progress	Specialist	2
9	case9705	19.11.09 10:32	Phone	Completed	Specialist	2
10	case3939	15.10.09 11:48	Mail	Registered	Specialist	2
11	case3939	15.10.09 11:48	Mail	Offered	Specialist	2
12	case3939	20.10.09 17:18	Mail	In progress	Specialist	2
13	case3939	20.10.09 17:19	Mail	At specialist	Specialist	2
14	case3939	21.10.09 14:49	Mail	In progress	Specialist	2
15	case3939	21.10.09 14:49	Mail	In progress	Specialist	2
16	case3939	28.10.09 10:17	Mail	In progress	Specialist	2
17	case3939	28.10.09 10:18	Mail	Completed	Specialist	2
18	case9704	20.10.09 14:19	Mail	Registered	1st line	0
19	case9704	20.10.09 14:24	Mail	Completed	1st line	0
20	case9703	20.10.09 14:40	Phone	Registered	1st line	0
21	case9703	20.10.09 14:58	Phone	Completed	1st line	0
22	case9702	24.8.09 12:24	Mail	Registered	2nd line	2
23	case9702	24.8.09 12:30	Mail	Offered	2nd line	2

Figure 2.3: An example of an event log

To illustrate the concept of process mining, we consider the process log shown in Table 2.3. This log contains the following information:

- Events - Each event corresponds to an activity that was executed in the process.
- Cases - Multiple events are linked together to make a process instance or case.
- Logically, each case forms a sequence of events ordered by their time-stamp.

The process mining approach needs the following minimum requirement from an event log as an input for effort extraction:

1. Case ID: A case identifier, also called process instance ID, is necessary to distinguish different executions of the same process.
2. Activity: There should be names for different process steps or status changes that were performed in the process. If you have only one entry (one row) for each process instance, then your data is not detailed enough.
3. Time-stamp: At least one time-stamp is needed to bring the events in the right order. Of course you also need timestamps to identify delays between activities and identify bottlenecks in your process.

[7]Cambridge Dictionary [8] defined process as a series of actions or steps taken in order to achieve a particular end.

For this thesis, a process is a list of events that occur in a particular class. Effort can be extracted, once the process is identified. Note that we did not use any particular Process Mining approaches, we only use the process mining concepts.

2.7 Related work on IDE log analysis for studies on software maintainability

In 2015, Minelli et al. aimed to record interaction data and measure the time effectively devoted to different activities (e.i. read, inspect, edit). With the recorded data, Minelli et al. aimed to provide insights on the distribution of development activities. To analysis how developers spend their time, Minelli et al. collected IDE activity logs. The dataset was collected for about 200 hours of development time which amounts to more than 5 million of IDE events. The contributions of this study were an inference model of development activities to precisely measure the time of different activities and a brief presentation of DFLOW, the tool with which we collect interaction data.

In 2016, Amann et al. performed a case study of how industrial C# developers use Visual Studio. Amann et al. developed FEEDBAG, a tool that anonymously captures developers IDE activity logs. The tool was deployed at an industry partners software development department, in which more than 400 developers write software in C#. The dataset collected consist of more than 3.5 million interaction events over a total of 6,300 work hours. The contributions of this study were an open-source tool, FEEDBAG, a case study of how professional C# developers use Visual Studio and a discussion of opportunities to advance the research in IDEs and developer-assistance tools.

2.7.1 Related work on effort extraction from IDE activity logs

In 2013, Sjøberg et al. researched the relationship between code smells and maintenance effort. An Eclipse plugin namely Mimec[36] was used to measure the exact amount of time a developer spent maintaining each file. There was no further description of how the amount of time spent maintaining each file was calculated.

In a follow-up study by Soh et al. [55], the effects of code smells at the activity level was investigated. By activities, Soh et al. [55] means reading, editing, searching, and navigating, which is performed independently over different files during maintenance. In this study, Soh et al. [55] provide a more detailed explanation of how Mimec was used for the effort extraction. However, Soh et al. [55] did not describe the accuracy of that process.

Chapter 3

Methodology

3.1 Overview

This chapter describes the conceptual and methodological framework used to accomplish the goal of this thesis. The red part of figure 3.1 depicts the log analysis tool (LAT), which will be described in detail in Section 3.2. The blue part of Figure 3.1 depicts the empirical study that answered the second research question, which will be described in detail in Section 3.3.

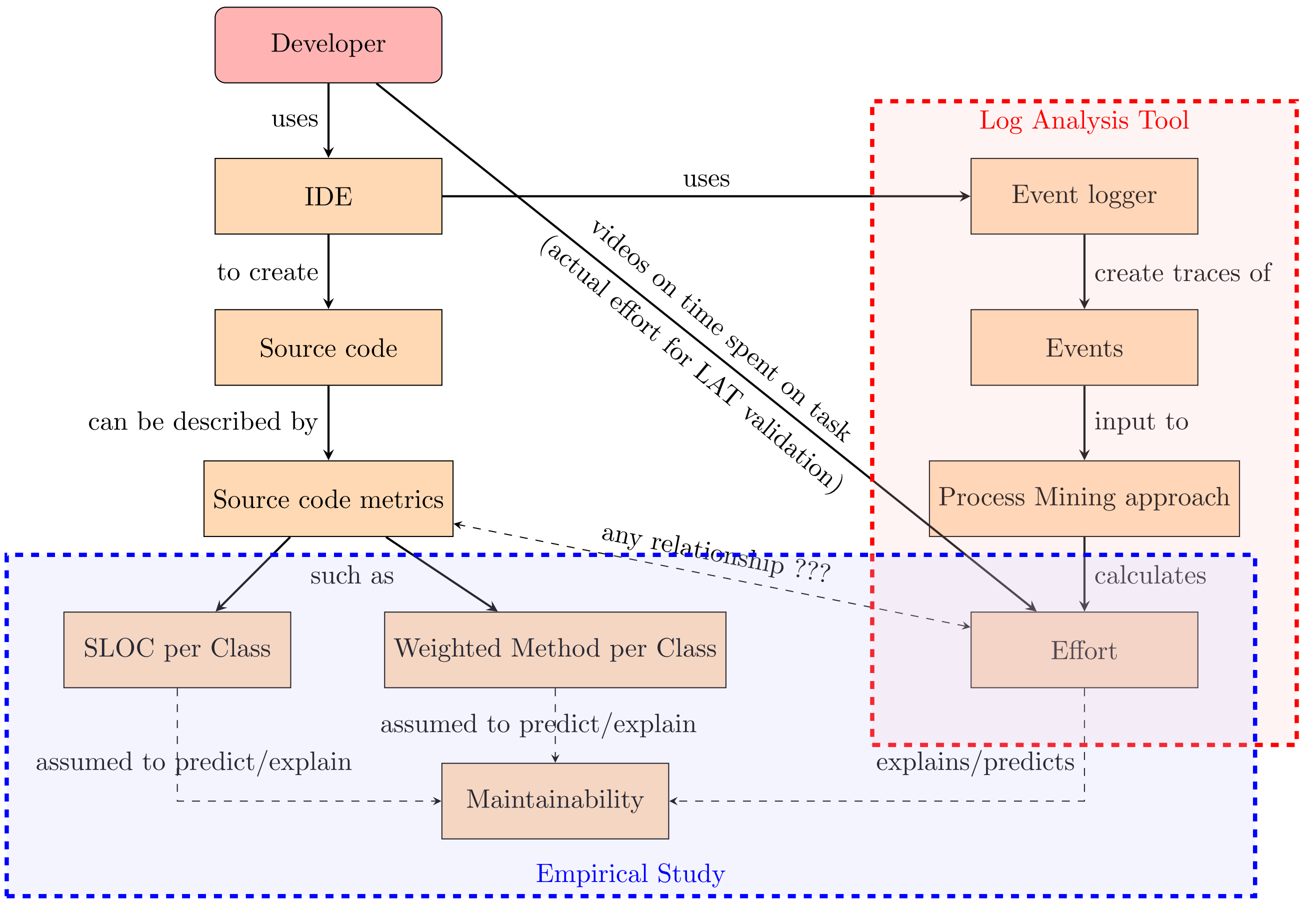


Figure 3.1: Conceptual and Methodological Framework used in the thesis.

3.2 Log analysis tool (LAT)

3.2.1 Effort extracting approach

This section describes the techniques and process used to implement LAT, including a description of the algorithm that extracts effort from IDE activity logs.

The process mining approach enables the extraction of effort from the IDE activity logs as explained in section 2.6. For example, consider the table 3.1 as a developer IDE activity logs. It contains the minimum requirement of attributes that enables the process mining approach. It has timestamps, activities denoted by kind and Case ID denoted by the class name. Below we illustrate the method of extracting effort using the example log in table 3.1.

Table 3.1: An example of a developer IDE activity logs

#	Time-stamp	Kind	ClassName
1	19-02-1991 12:00:00	open	User.java
2	19-02-1991 12:05:00	read	User.java
3	19-02-1991 12:10:00	scroll	User.java
4	19-02-1991 12:15:00	edit	User.java
5	19-02-1991 12:20:00	open	Auth.java
6	19-02-1991 12:25:00	edit	Auth.java
7	19-02-1991 12:30:00	scroll	Auth.java
8	19-02-1991 12:35:00	open	User.java
9	19-02-1991 12:40:00	read	User.java
10	19-02-1991 12:51:00	scroll	User.java

The approach is as follows: we create process cases where the class names group the events.

1. For each event, we calculate the elapsed time. The elapsed time (ET) is the start time (ST) of the log minus the current time (CT) of the event ($ET = CT - ST$).

2. We calculate the accumulated time (AC) of each process case by adding all the events' elapsed time ($AC = \sum_{i=ET}^{N} i$).

3. Finally, all idle times (IT) are removed from the accumulated time. An example of idle time is when a user is programming, she/he leaves her desk for a pause while everything continues to run. To detect it, Minelli et al. [42] proposed to set a "minimum idle time", of 10 minutes by default. If the difference between two events is more than 10 minutes, then the difference is also considered as idle time. Listing 3.5 depicts the implementation of removing idle time.

For further explanation, each time a class opens, the elapsed time is 0 minute. After that, the elapsed time is calculated from the previous event. For example, in table 3.1, the elapsed time of the second event is 5 (12:05:00 - 12:00:00) minutes. Table 3.1 also shows an example of an idle time. The elapsed time of the last event is 11 (12:51:00 - 12:40:00) minutes. This elapsed time is more than the "minimum idle time," of 10 minutes. Hence it will be subtracted from the accumulated time. See Table 3.2 for the full solution.

User.java			
#	**Elapsed (mins)**	**Idle (mins)**	**Total**
1	0		
2	5		
3	5		
4	5		
8	0		
9	5		
10	11	11	
AC	31	11	20

Auth.java			
#	**Elapsed (mins)**	**Idle (mins)**	**Total**
5	0		
6	5		
7	5		
AC	10		10

Table 3.2: An example of how effort is extracted from Table 3.1

3.2.2 Architectural design of LAT

The design overview contains two parts: Loggers and the LAT itself as presented in Figure 3.2. The blue part of Figure 3.2 shows three examples of loggers (Mylyn, FLUORITE, and Mimec). More information about loggers is in the background and related work chapter. The red part of Figure 3.2 shows the decomposition of LAT. It contains the utils, the model and the user interface.

The utils package contains classes that parse the event logs into process models. For LAT to support the analysis of different types or formats of IDE activity logs, an abstraction was made on top of more implementation/format related details. Each logger has its separate class for that process because each event log file could be structured differently. For example, Mylyn saves the events to either a CSV or XML file; FLUORITE saves the events to XML files and Mimec saves the events to a CSV file.

The process model package contains the process mining approach that was explained earlier. The package has process cases, process case, and events. The process cases are just a map containing the class names with their respective process case. Listing 3.4 illustrates an example of process cases. Listing 3.2 illustrates an example of a process case. A process case contains all the events that belong to a class. Finally, Listing 3.3 shows an example of the class definition of events.

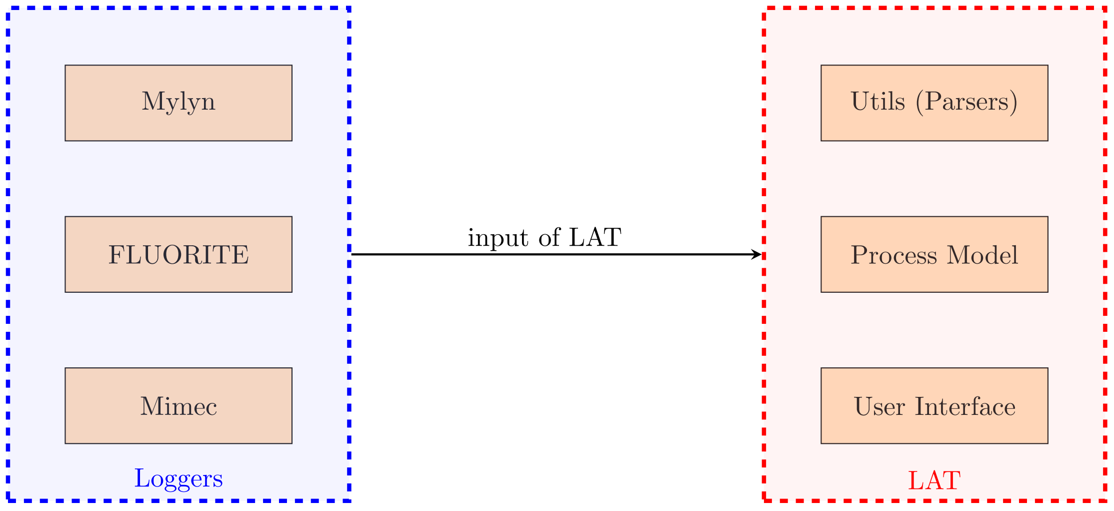


Figure 3.2: LAT Architectural Design

```java
package edu.cwi.espionage.model;

import java.util.ArrayList;

public class ProcessCase implements Comparable<ProcessCase>{
  private String caseId;
  private List<Event> events;
  private long startTime;
  private long idleTime;
  private Event lastEvent;
  private IdleTimeTable idleTimeTable;

  public ProcessCase(String caseId) {
    this.events = new ArrayList<Event>();
    this.caseId = caseId;
    this.idleTimeTable = new IdleTimeTable();
  }
```

Listing 3.2: Example of ProcessCase class definition in Java

```java
package edu.cwi.espionage.model;

import java.util.Date;

public class Event implements Comparable<Event>{

  private Date timestamp;
  private String activity;
  private long elapstime;
  private String caseId;

  public Event(Date timestamp, long elapstime, String activity) {
    this.timestamp = timestamp;
    this.activity = activity;
    this.elapstime = elapstime;
  }
```

Listing 3.3: Example of Event class definition in Java

```java
Map<String, ProcessCase> processCases = new HashMap<String, ProcessCase>();
```

Listing 3.4: Example of ProcessCases code in Java

```java
private static final long MINIMUM_IDLE_TIME = 600; //10 minutes as default.

private Long getElapsedTime(Event firstEvent, Event secondEvent){
  return Math.abs(secondEvent.getTimestamp().getTime() - firstEvent.getTimestamp().getTime());
}

private Boolean IsInactive(Event firstEvent, Event secondEvent){
  boolean isInactive = false;
  long elapsedTime = getElapsedTime(firstEvent,secondEvent);
  long elapsedTimeSeconds = elapsedTime/1000;

  if(elapsedTimeSeconds > MINIMUM_IDLE_TIME){
    isInactive  = true;
  }
  return isInactive;
}

public Long calculateIdleInactiveTime(Event e1, Event e2) {
  Event firstEvent  = e1;
  Event secondEvent = e2;

  if(e1.getTimestamp().after(e2.getTimestamp())){
    firstEvent  = e2;
    secondEvent = e1;
  }

  if (IsInactive(firstEvent, secondEvent)) {
    return secondEvent.getElapstime();
  }

  return new Long(0);
}
```
Listing 3.5: A code snippet of extracting Idle time code in Java

Figure 3.3 presents the user interface (UI) of LAT. Figure 3.3 has three parts: the filters, the classes, and a line graph. The filters allow the user to filter the data. For example, extracting effort spent maintaining a class on a particular day and range of hours. Below the filters are the classes within their respective packages. Finally, the line graph shows maintenance effort per day for a class upon selection. LAT also saves effort (time in minutes) on a class level in a comma-separated value (CSV) file as it is illustrated in table 3.3. The full code can be found at http://bit.ly/2wiyTIS.

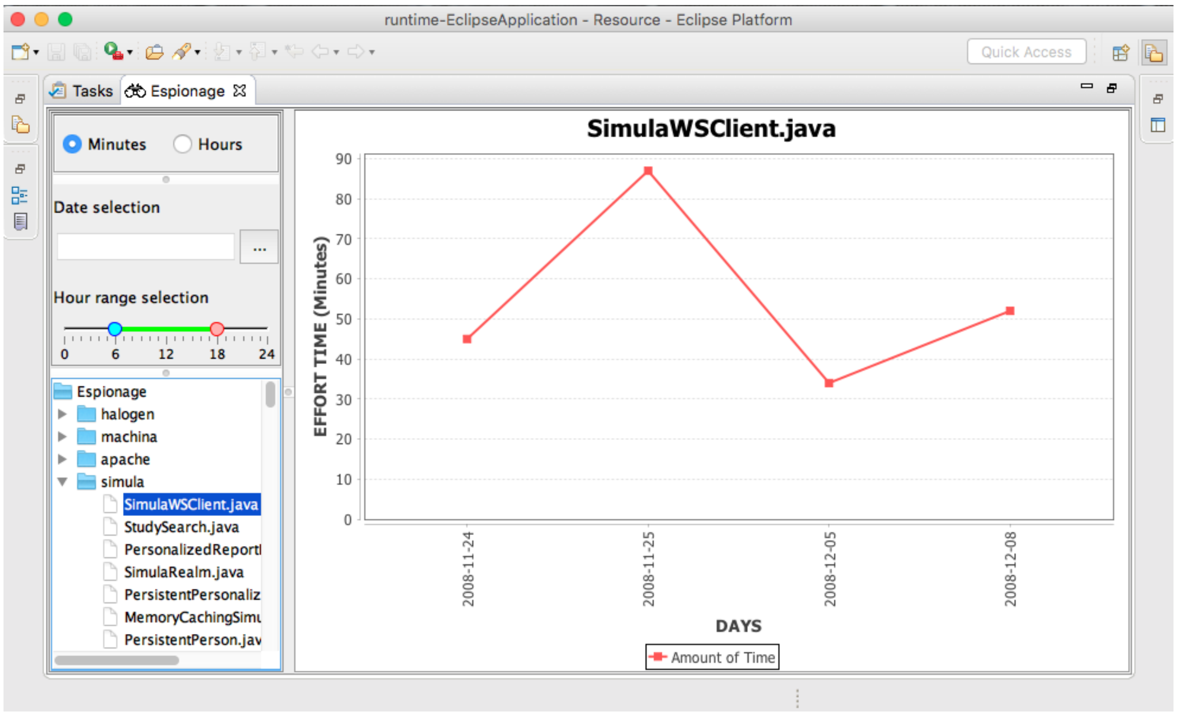


Figure 3.3: An example of LAT UI

Table 3.3: Example of LAT saved data.

Project	EntityName	Time(milli)	Time(mins)
gjt	SimulaWSClient.java	3558000	59
gjt	SimulaWS.java	918000	15
gjt	Driver.java	231000	3
halogen	LoggableStatement.java	767000	12
halogen	TableRenderer.java	95000	1
halogen	TableTag.java	27000	0
halogen	Table.java	8000	0
machina	Exercises.java	8486000	141
machina	DB.java	231000	3
machina	SimulaWSClient.java	188000	3
machina	ReportServlet.java	70000	1
machina	DBTest.java	29000	0
apache	ServletServer.java	611000	10
apache	SimulaWSClient.java	223000	3

3.2.3 Evaluation

There are many issues to measuring effort from logs according to previous work. Sjøberg et al. [52] explained that it has been historically difficult to verify if log-based effort extractions are accurate because of the lack of grounded truth or empirical measurements of effort during a sufficiently extended period. Normally, reported experiments only last a couple of hours[13]. Deligiannis et al. [13] is an example, where it only lasted for one hour and a half. Soh et al. [54] also noted that IDE event logs contain noise. Noises can be 0 duration events and overlapping events.

Before using LAT, it is important to verify whether the efforts extracted from the event logs are accurate. The accuracy of LAT is important because it will validate effort extraction from event logs and it will enable higher confidence on further results interpretations. Thus, the evaluation reported in this section intends to answer the following research question:

> *How accurate can file-level effort measurements be when based on IDE activity logs analysis?*

Evaluation Approach

Two different data sets were used to evaluate the accuracy of LAT. The data sets are from a pilot evaluation conducted by the author of the thesis and the replication package from the study by Soh et al. [54]. Both data sets were built by two different means, as presented in Figure 3.4. On one side, an Eclipse tool is used to gather raw interaction data, and on the other side, software video players such as VLC and QuickTime were used to capture video of developers' screens while completing a task. The blue part (Dataset A) in Figure 3.4 shows how LAT extracts effort from IDE activity logs (Expected Effort). The red part (Dataset B) in Figure 3.4 shows the real effort or ground truth (Actual Effort) that was spent on each file, which is extracted by manually examining the recorded videos. Then, we compare the expected effort with the actual effort to measure the accuracy of LAT. The analysis methods used to compare compare the degree of similarity or consistency between the two data sets (and thus, the accuracy of LAT) are regression analysis [20], cosine similarity [25, p. 77] and a Bland-Altman plot [62]. Each of these analyses methods will be explained in the latter text.

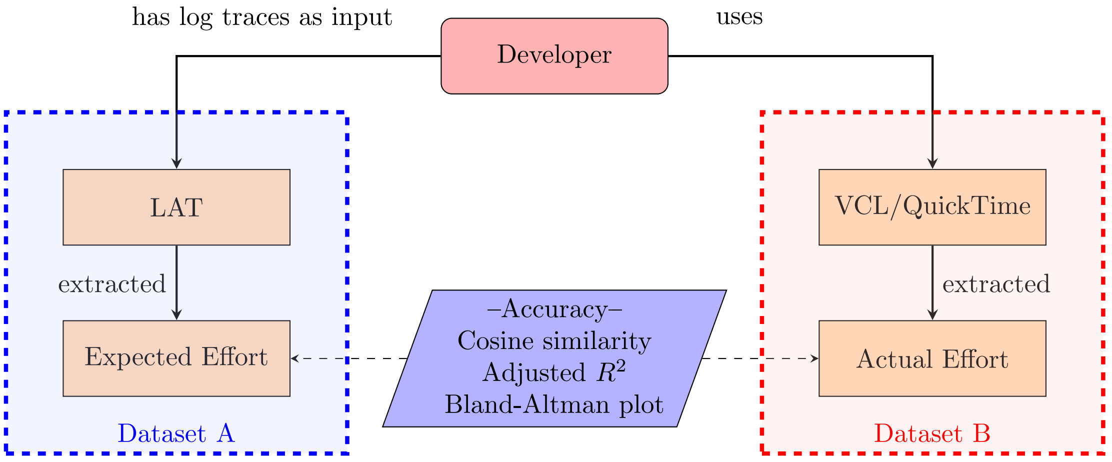


Figure 3.4: LAT Evaluation Overview

As mentioned previously, during the evaluation of the logging tools, it was possible to observe that most of the logging tools presented in the background chapter were not available anymore at the time of this study. FLUORITE logging tool was found as the best fit for this study because it was available and records the data required to extract effort at the file level. However, in principle, any available datasets from any event logging tool can be used for effort extraction, provided that the datasets fulfill the requirements explained in Section 2.6 (e.i. Process Mining). The dataset used in this study can be found at http://bit.ly/2vE9F77.

Pilot experiment: We conducted a pilot evaluation with FLUORITE that lasted about 100 minutes. Two participants performed their daily programming tasks using the Eclipse IDE together with the FLUORITE plugin. We collected their IDE activity logs at the end. We also captured video recordings of the participants screens using QuickTime.

Mylyn dataset: We used a dataset from Soh et al. [54] study about "Noises in Interaction Traces Data and their Impact on Previous Research Studies." We used 175 minutes of data consisting of IDE activity logs with their respective videos.

For both datasets, LAT was used to calculate the developer's effort. The tool calculates and saves the effort spent (in minutes) on a file level in a comma-separated value (CSV) file as illustrated in Figure 3.3. We also transcribed the videos manually to extract the actual effort that was spent on each file.

Analysis conducted for evaluating the accuracy of LAT: Before applying the aforementioned analysis methods, we first analysed the distributions in our data using histograms, since these analyses perform best when measurements are normally distributed [23, 35, 55]. To understand the distribution, we use the skewness method. According to Bai and Ng [4], measurements are normally distributed when skewness is 0. If it is above 0, then it is positively skewed. If it is below 0, then it is negatively skewed. In case the measurements are not normally distributed, a logarithmic transformation of original data can be applied [23, 35, 55]. The subsequent section describes the three aforementioned analysis methods.

Cosine similarity: Han et al. [25, p. 77] defined Cosine similarity as follows:

> *Cosine similarity is a measure of similarity that can be used to compare documents or, say, give a ranking of documents with respect to a given vector of query words. Let x and y be two vectors for comparison. Using the cosine measure as a where $\|x\|$ is the Euclidean norm of vector $x = (x_1, x_2, ..., x_p)$, defined as $\sqrt{x_1^2 + x_2^2 + ... + x_p^2}$. Conceptually, it is the length of the vector. Similarly, $\|y\|$ is the Euclidean norm of vector y. The measure computes the cosine of the angle between vectors x and y. A cosine value of 0 means that the two vectors are at 90 degrees to each other (orthogonal) and have no match. The closer the cosine value to 1, the smaller the angle and the greater the match between vectors.*

Regression analysis: Field and Miles [20] defined regression analysis as follows:

> *Regression analysis is a way of predicting an outcome variable from one predictor variable (simple regression) or several predictor variables (multiple regression).*

In the regression analysis, We will be focusing at the adjusted square of Pearson correlation which is called the adjusted coefficient of determination (Adjusted R^2) and the standard error measures. The adjusted R^2 indicates the percentage of the dependent variable that can be explained by the predictor variable. The standard error measures the accuracy of the dependent variable's coefficient by estimating the variation of the coefficient if the same test were run on a different sample of our population.

Bland-Altman plot: Zaki et al. [62] explained Bland-Altman plot as follows:

> *Bland and Altman assess the agreement between two quantitative methods of measurement. The formula for the limits of agreement is given as: Limits of Agreement = mean difference 1.96 x (standard deviation of differences). The limits of agreement are dependent on the assumptions that the mean and standard deviation of the differences are constant throughout the range of measurement, and that the distribution of these differences approximately follows a normal distribution. Bland and Altman proposed a scatter plot of the difference of two measurements against the average of the two measurements and a histogram of the differences to check this assumption.*

The above mentioned method gained momentum in the clinical laboratory research (medicine) as an alternative to correlation and linear regression when changing one method for another one or for evaluating a new or alternative method [62, 23, 2]. Correlation and linear regression only measure how strongly, pairs of variables are related but not the level of agreement [62, 23, 2]. For example given two sets of data X and Y, where $Y = 2X$. The two sets of data are highly correlated and fit a perfect line. However, it is obvious that the value of Y is twice the value of X (i.e. no agreement).

Results and discussions

Figure 3.5 and 3.6 show the distributions and skewness of actual effort and expected. Figure 3.5 and 3.6 show skewed distributions with long tails. 3.11 and 3.16 are the skewness respectively. Hence a log transformation is required for further analysis because the measurements are not distributed normally.

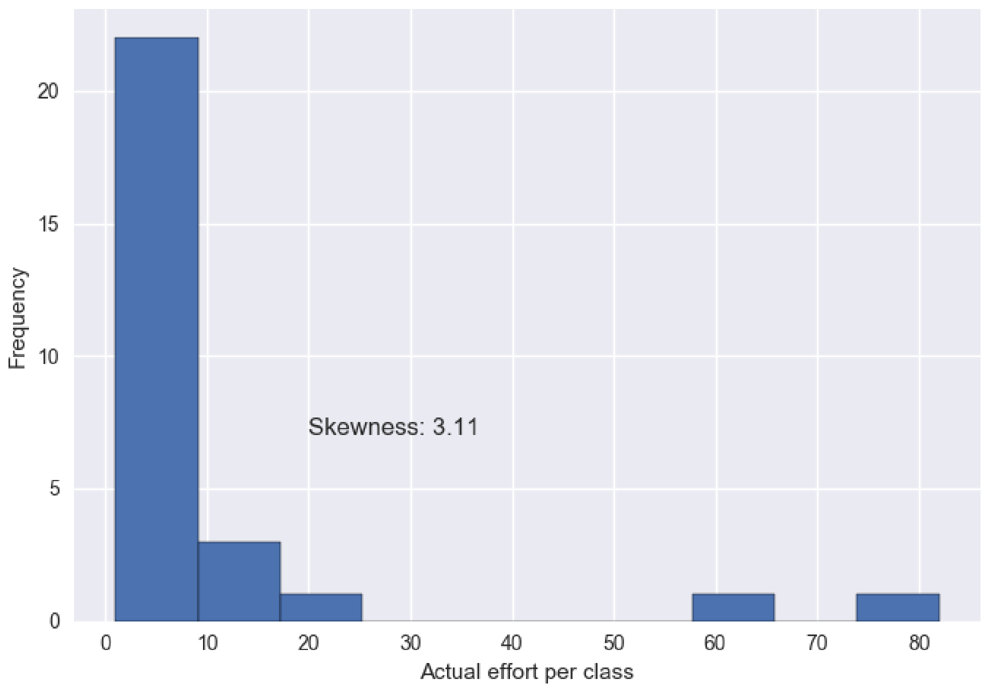


Figure 3.5: Actual effort distribution

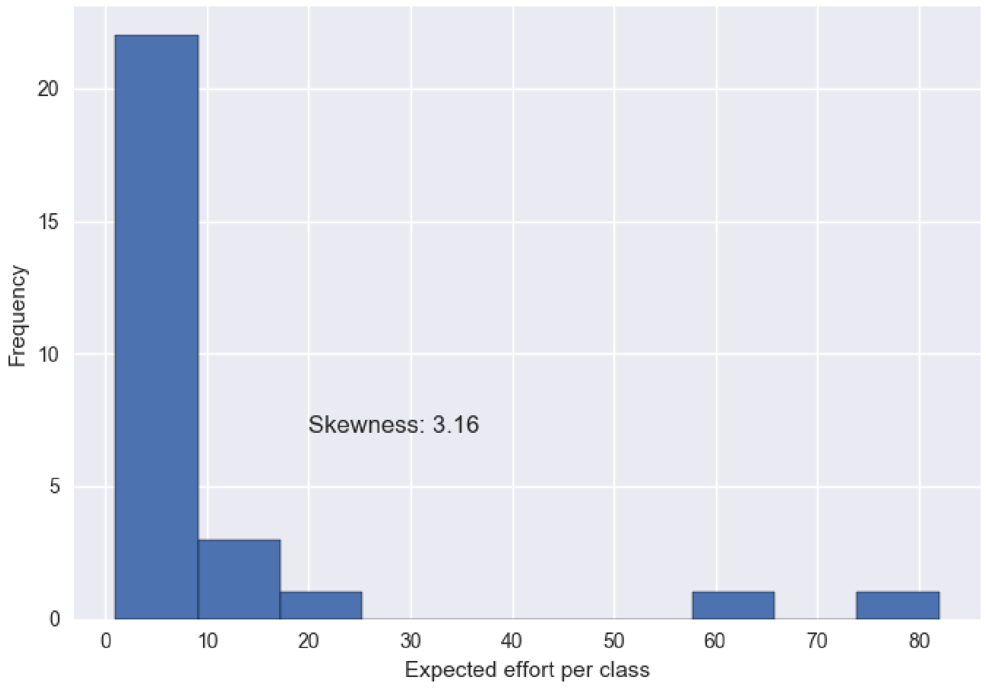
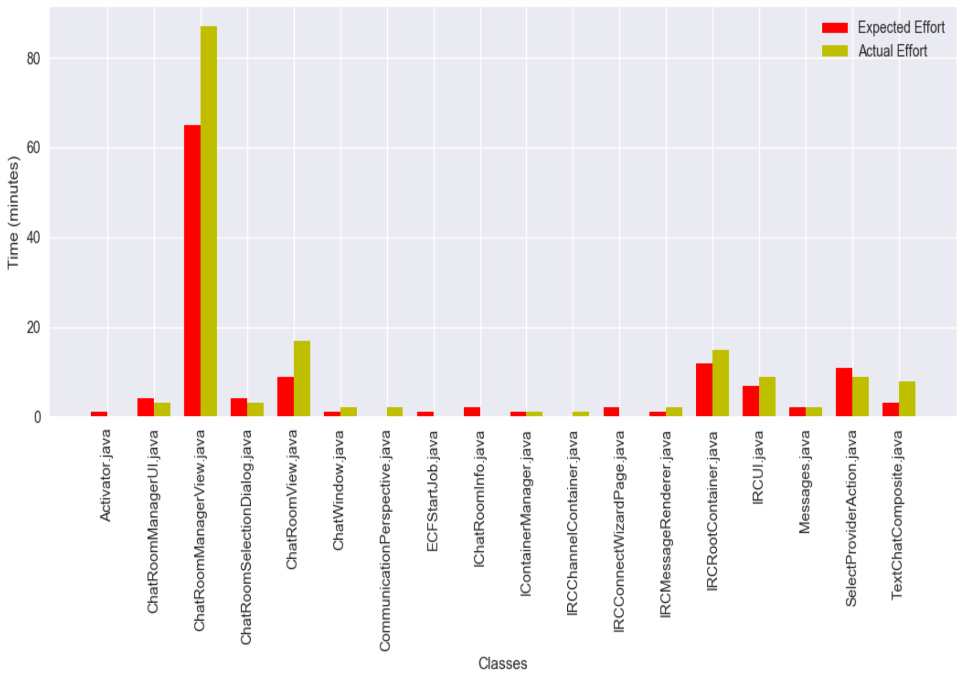


Figure 3.6: Expected effort distribution

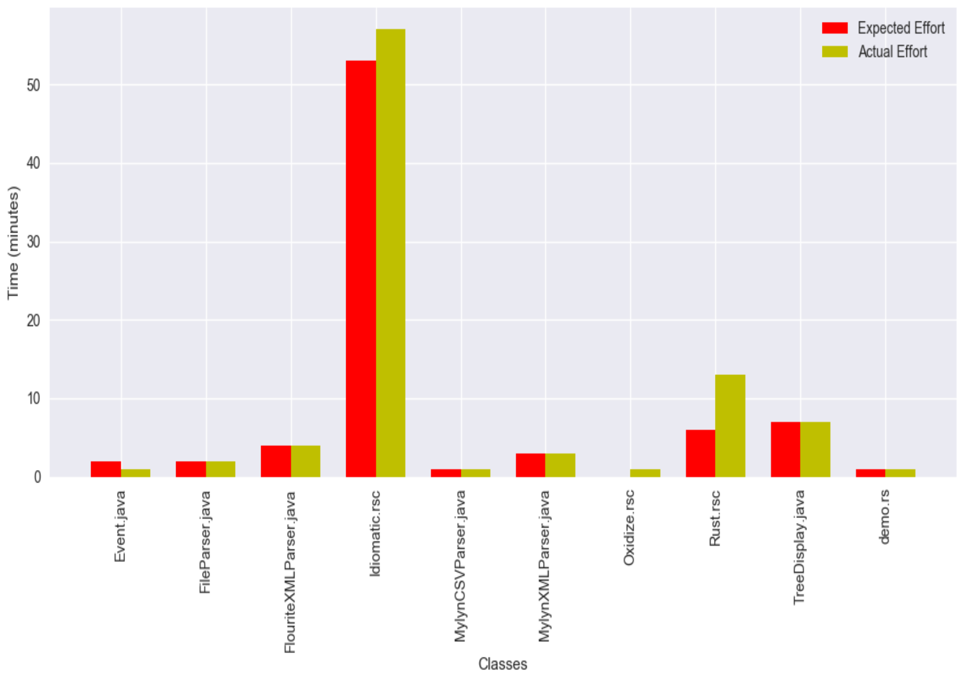
Figure 3.7 presents the differences in the effort devoted to each file while performing the tasks, between Eclipse extracted effort and real effort from the recorded videos. Figure 3.7a illustrates Mylyn dataset and Figure 3.7b illustrates Fluorite dataset with recorded videos respectively.

Finding 1: The cosine similarity between the actual effort and expected effort is 0.87

Figure 3.7b shows that the extracted effort is almost identical to the real effort from the video. Figure 3.7a shows more discrepancy. To evaluate the results, cosine similarity of the datasets are calculated. Finding 1 showed a cosine similarity of 0.87 between the datasets. From the definition presented above, the value 0.87 is close to 1. Therefore, the datasets are considered quite similar.



(a) Mylyn



(b) Fluorite

Figure 3.7: Difference between Eclipse extracted effort (Expected Effort) and real effort from video (Actual Effort) in times spent in each file

In Table 3.4 we can observe that, in accordance to the correlation ranking provided by Rumsey [48], expected effort is strongly related to the actual effort of developers, with an adjusted R^2 of 75%, which is also significant at $p < .001$. This indicates that 75% of actual effort can be explained by our predictor variable, expected effort.

Finding 2: Actual and expected effort have a strong positive correlation ($adj.R^2 = .75$, $N = 28$, $p < .001$).

Table 3.4: Results of regression analysis for LAT accuracy

OLS Regression Results

Dep. Variable:	actual	R-squared:	0.754
Model:	OLS	Adj. R-squared:	0.745
Method:	Least Squares	F-statistic:	79.9
Date:	Mon, 21 Aug 2017	Prob (F-statistic):	0.001
Time:	15:08:03	Log-Likelihood:	-20.068
No. Observations:	28	AIC:	44.14
Df Residuals:	26	BIC:	46.8
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	$P > \|t\|$	[95.0% Conf. Int.]	
Intercept	5.55E-14	0.097	5.71E-16	1.000	-0.200	0.200
expected	0.8686	0.097	8.939	0.001	0.669	1.068

Table 3.5 shows expected and actual effort after log transformation, from which allows to construct the Bland-Altman plot and hence, to evaluate the level of agreement between the measurements. In the first column, a series of expected effort is shown, obtained by LAT. The second column shows the actual effort obtained from video recordings. Therefore, each line shows paired data. Column 4 shows the differences. An ideal model would claim that the measurements obtained by LAT or video recording gave the same results. So, all the differences would be equal to zero. However, that is not the case. Figure 3.8 shows a small degree of error. Figure 3.8 shows a scatter plot XY, in which the Y-axis shows the difference between the two paired measurements (expected effort - actual effort) and the X-axis represents the average of these measures ((expected effort + actual effort)/2). In other words, the difference between the two paired measurements is plotted against the mean of the two measurements. The Bland-Altman plots 95% of the data points within ±1.96 standard deviation of the mean difference. The Bland-Altman plot does not say if the agreement is sufficient or suitable, it is up for personal interpretation. With that in mind, examining Figure 3.8, there is a small bias, but the 95% limits of agreement can be accepted for the goal of this thesis. Figure 3.7 also depicts the differences in the two methods. The expected effort is more often than not underestimated. There is a few reason for these differences: Mylyn IDE activity logs sometimes miss the recording of some events, the manual transcript of the videos could suffer from human error and the idle time measure of 10 minutes can be considered rather subjective (or arbitrary).

Table 3.5: Data of an agreement between two methods (Expected and Actual effort).

log(expected effort)	**log(actual effort)**	**Mean (expected+actual)/2**	**Difference (expected-actual)**
1.00	0.00	0.50	1.00
2.32	2.00	2.16	0.32
6.04	6.38	6.21	-0.33
2.32	2.00	2.16	0.32
3.32	4.25	3.78	-0.93
1.00	2.00	1.50	-1.00
0.00	1.58	0.79	-1.58
1.00	0.00	0.50	1.00
1.58	1.58	1.58	0.00
1.58	1.58	1.58	0.00
2.32	2.32	2.32	0.00
1.58	0.00	0.79	1.58
1.00	1.00	1.00	0.00
0.00	1.00	0.50	-1.00
1.58	0.00	0.79	1.58
1.00	1.00	1.00	0.00
3.70	4.17	3.94	-0.47
3.00	3.32	3.16	-0.32
5.75	5.88	5.82	-0.13
1.58	1.58	1.58	0.00
1.00	1.00	1.00	0.00
2.00	1.58	1.79	0.42
0.00	1.58	0.79	-1.58
2.81	3.91	3.36	-1.10
3.58	3.46	3.52	0.13
2.00	3.32	2.66	-1.32
3.00	3.17	3.08	-0.17
1.00	1.58	1.29	-0.58
mean			-0.15
standard deviation			0.83

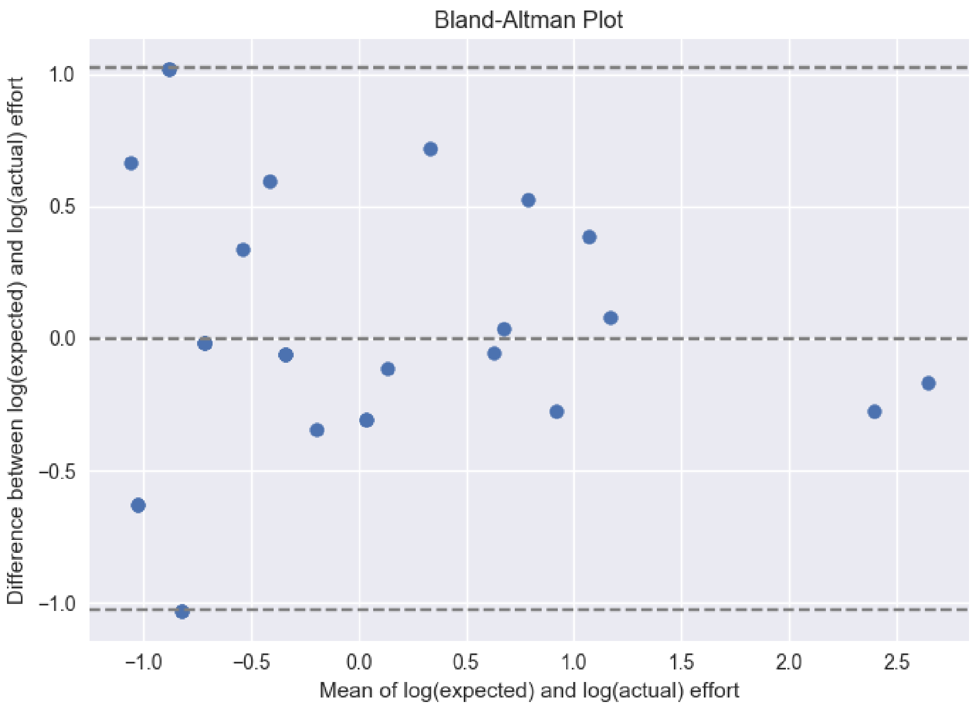


Figure 3.8: Results of the Bland-Altman Plot

Conclusion

There have been many problems in measuring effort from logs. Problems such as 0 duration and overlapping events have been reported. This chapter uses video recordings to measure the accuracy of LAT. We used cosine similarity, adjusted R^2 and Bland-Altman Plot as methods for validating LAT. The cosine similarity showed a similarity index of 87%. The adjusted R^2 produced a result of 75%. Finally, the Bland-Altman Plot showed only a small degree of bias in the differences between the two methods. This evidence suggests that LAT can potentially be used for further analysis of effort extraction from IDE activity logs.

3.3 Empirical study

This section describes the study that was used to answer research question 2: *Can we effectively use previously measured SLOC and CC of a class to estimate/predict future maintenance effort on source code at class level?*.

3.3.1 Hypothesis

As explained by Visser et al. [58, Chap. 6], small size classes provide a direct path toward loose coupling between classes. Loose coupling means that the class level design will be much more flexible to facilitate future changes. By ”flexibility” Visser et al. [58, Chap. 6] means that changes can be made by limiting unexpected effects of those changes. This means that small size classes allow developers to work on an isolated part of a codebase. Hence, the larger the class, the higher the effort needed for maintainability. Based on Visser et al. [58, Chap. 6] above information, we have the following hypothesis.

Hypothesis 1 *There is strong linear correlation between total SLOC and effort metrics at a file level.*

Visser et al. [58, Chap. 3] explained that a simple unit is easier to understand, and thus modify, than a complex one. At a file level, WMC is used to measure the overall complexity of a class. Rosenberg et al. [47] noted that a class with a low WMC usually points to greater polymorphism. A class with a high WMC tends to be complex and therefore harder to reuse and maintain. Based on Visser et al. [58], Rosenberg et al. [47, Chap. 3] above information, we have the following working hypothesis.

Hypothesis 2 *There is strong linear correlation between total WMC and effort metrics at a file level.*

3.3.2 Research method

To evaluate the effect of CC and SLOC on the effort required during maintenance activities, we answer the following research question: *Can we effectively use previously measured SLOC, CC to estimate/predict future maintenance effort?* To do this, we used the same modeling analysis (e.i. histograms, descriptive statistics and scatter plots) from Landman et al. [35]'s study to understand the data. Then, we use Pearson correlation to answer hypothesis 1 and 2. Finally, we use stepwise regression just as Soh et al. [55] to assess how SLOC and CC could explain maintenance effort.

Distributions: Before answering the research question, we describe the distributions in our data using histograms and descriptive statistics (median, mean, min and max). Landman et al. [35] explained how the shape of distributions can have an impact on the correlation measures used. It should be noted that to fit best a linear model, the variables should be close to a normal distribution. We apply log transformation method to compensate for the skewness of the distribution just as it was in the evaluation of LAT. This previous studies [55, 35, 19, 30, 27] used the log transformation method as well.

Scatter plots: A scatter plot matrix will be used to illustrate the relationship between total SLOC, WMC and effort.

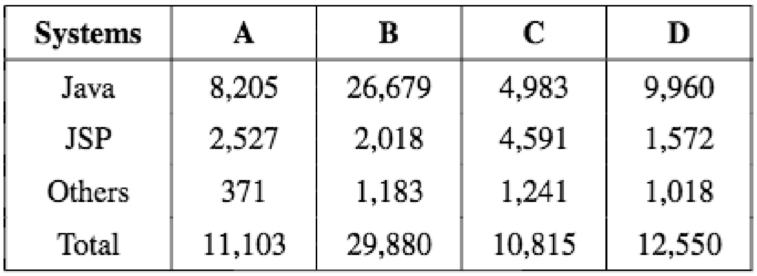
Correlation: We use Pearson correlation to measure the degree of linear relationship between two variables. The square of Pearson correlation is called the coefficient of determination (R^2).R^2 estimates the variance in the power of one variable to predict the other using linear regression. According to Rumsey [48], if the R^2 is more than .70 then we will accept hypothesis 1 and 2.

Type	**Variable**
Independent	Static Metrics (SLOC, CC)
Dependent	Effort (minutes spend on maintaining a file)
Control	Developer and Round

Table 3.6: Variables involved in the study

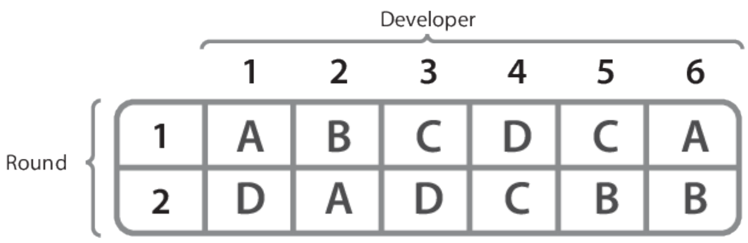
We used stepwise regression to assess how SLOC and CC could explain maintenance effort. We build a model for each independent variables. We started with Model 0 where SLOC is used to explain maintenance effort. In Model 1, we add the WMC to Model 0 to measure how WMC contributes to explaining maintenance effort. We put NA (Not Applicable) in case a variable is not used in a model. By incrementally adding variables to the model, we assess the contribution of each variable to explaining maintenance effort. Table 3.6 further illustrates variables that will be involved in the study to estimate maintenance effort.

Dataset



Systems	A	B	C	D
Java	8,205	26,679	4,983	9,960
JSP	2,527	2,018	4,591	1,572
Others	371	1,183	1,241	1,018
Total	11,103	29,880	10,815	12,550

Figure 3.9: SLOC Per file type for all four systems.
Soh et al. [55]



Round	Developer 1	2	3	4	5	6
1	A	B	C	D	C	A
2	D	A	D	C	B	B

Figure 3.10: Assignment of systems to developers in the case study rounds.
Soh et al. [55]

To perform the study in this research, we use the same dataset from two previous studies from Soh et al. [55] and Sjøberg et al. [52] in which the impact of code smell on maintenance effort was studied. Both of these studies used the same dataset. However, the difference is that Sjøberg et al. [52] measured sheer effort and Soh et al. [55] measured effort by type of activity. This dataset was acquired from Soh et al. [55]. Sjøberg et al. [52] generated this dataset by the following means: Six professional developers were hired to perform three maintenance tasks on four functionally equivalent Java Systems. Each developer performed two rounds of maintenance tasks as seen in figure 3.10. During maintenance task, Mimec [36] was used to record the IDE activity logs.

Measuring effort

We ran LAT on the IDE activity logs from Soh et al. [55] and Sjøberg et al. [52]. The tool calculates effort (in minutes) at file level and stores it in a comma-separated value (CSV) file as it is illustrated in table 3.3.

Measuring SLOC and CC

To precisely control the definition of both SLOC and CC, we use the same method as performed in Landman et al. [35] study. The the M3 framework from Izmaylova et al. [29] was used, which is based on the Eclipse JDT[1], to parse the full Java source code. To compute Source lines of code, a grammar in RASCAL[2] was defined to tokenize Java input into newlines, whitespace, comments and other words. The parser produces a list of these tokens which we filter to find the lines of code that contain anything else but whitespace or comments. SLOC and CC are stored in a comma-separated value (CSV) file as it is illustrated in table 3.7.

[1]http://www.eclipse.org/jdt
[2]Klint et al. [34] provide more information about Rascal

(a) SLOC

name	**sloc**
Formatter.java	106
OptionHandler.java	4
CategoryNode.java	100
Roller.java	62
AbsoluteTimeDateFormat.java	65
StudyDAO.java	971
DeleteStudyHandler.java	20
FindPublicationsCommand.java	22
AdminPrivilegesBean.java	18
ReloadingPropertyConfigurator.java	13
UserReportCommand.java	19
CategoryNodeEditorRenderer.java	19
ListVsVector.java	61
Logger.java	29

(b) CC

name	**cc**
LocationInfo.java	2
NewVsSetLen.java	3
LoadXMLAction.java	1
Loader.java	5
LogTableColumn.java	2
LogLevel.java	3
NTEventLogAppender.java	1
LogLog.java	3
LogManager.java	7
Parser.java	2
Parser.java	2
Parser.java	5
Parser.java	1
Parser.java	5

Table 3.7: Example of a) SLOC and b) CC measurements.

Mapping SLOC and CC to the effort data

To analyze the SLOC, CC and effort data, they are merged into a single dataset. Each line in table 3.8, represents a file. Each line contains the file name (file), total SLOC, a Weighted Methods per Class (WMC) and the effort. The data consists of 680 files in total.

Table 3.8: Example of the dataset saved after merging 3.3 and 3.7.

file	**sloc**	**wmc**	**effort**
Driver.java	9	1	3
PropertyConfigurator.java	271	44	1
StudyDAO.java	971	132	628
PeopleDAO.java	261	48	361
PublicationDAO.java	179	37	255
ViewUserReportHandler.java	34	5	118
WebConstants.java	72	1	101
StudyListSortTag.java	36	4	84
DAO.java	99	20	77

Chapter 4

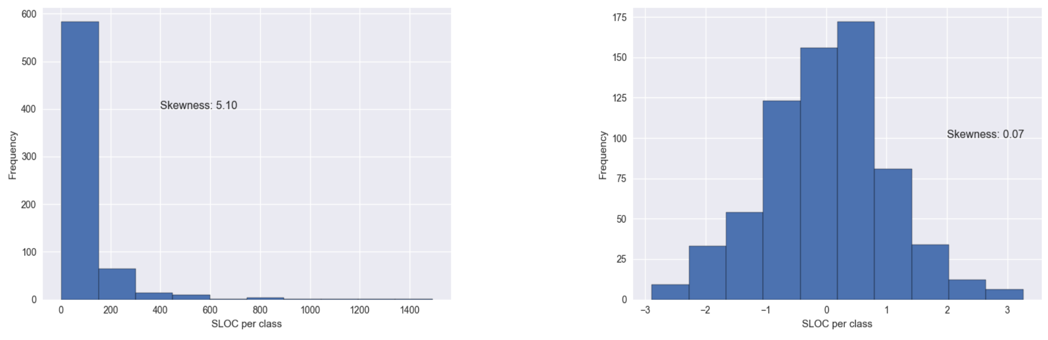
Results and discussion

In this chapter, we present the results and discussion of the empirical study described in Section 3.3. The first section contains the results and the second section presents the discussion concerning the results.

4.1 Results

Distributions

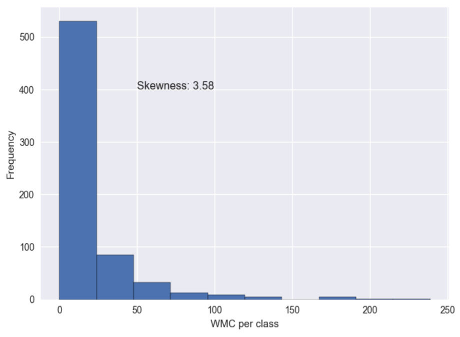
Figures 4.1, 4.2 and 4.3 show the distribution of total SLOC, CC and effort per class. Table 4.1 describes their distributions. Figure 4.1a, 4.2a and 4.3a show skewed distributions with a long tail. The skewness is 5.10, 3.58 and 5.15 respectively. According to Bai and Ng [4], skewness greater than zero shows large skewed distribution. After log transformation of SLOC, CC and effort, figure 4.1b, 4.2b and 4.3b showed a skewness of 0.10, -0.08 and 0.27 respectively.



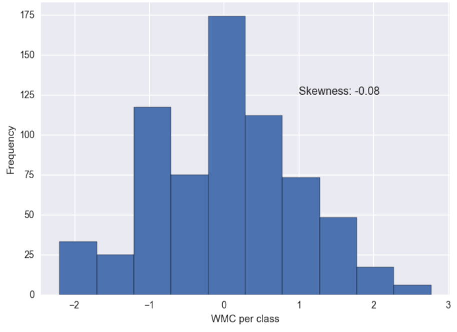
(a) Raw

(b) After log transformation

Figure 4.1: Histogram of total SLOC per class.

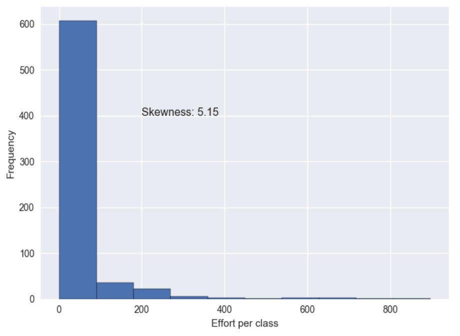


(a) Raw

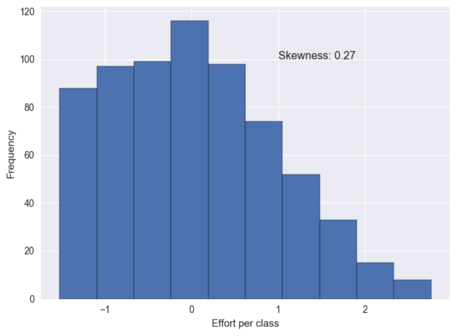


(b) After log transformation

Figure 4.2: Histogram of Weighted Methods per Class (WMC).



(a) Raw



(b) After log transformation

Figure 4.3: Histogram of effort spent per class.

Table 4.1: Descriptive statistic of total SLOC, WMC and effort per file .

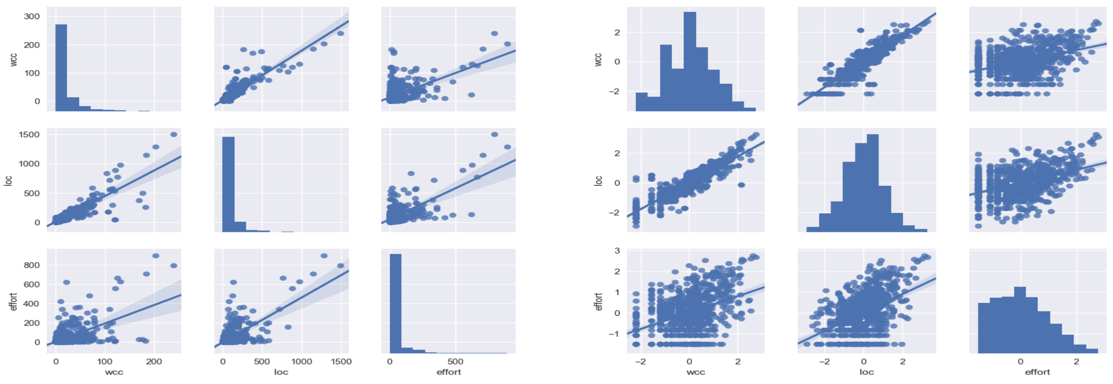
Variable	Min.	Max.	Mean	Median
SLOC	3	1492	94	56
WMC	0	239	19	11
Effort	1	897	40	11

Scatter plots

Figure 4.4 illustrates graphs of the relationships between SLOC, CC and effort. Figure 4.4b shows a graph of the relationship with the raw data. The raw plot shows a widely scattered and noisy field, with a high concentration of points in the left corner. At first glance, the relationship between the variables seems to be weak. Figure 4.4b shows a graph of the relationship after power transformation of the raw data. The power transformation plot also shows a widely scattered and noisy field. It appears that there is no relationship between the variables.

Pearson correlation

Pearson's correlation was run to determine the relationship between 680 files' SLOC, WMC and effort values. There was a weak positive correlation between SLOC and effort $adj.R^2 = .21$, $N = 680$, $p < .001$). A 0.21 adjusted R^2 presented in figure 4.2 indicates 21% of effort can be explained by



(a) Raw (b) After log transformation

Figure 4.4: Scatter plot of total SLOC, CC and effort per file.

our predictor variable, SLOC. According to the p-value (0.001) with confidence interval of 95%, this result is significant because it is lower than 0.5.

There was a weak positive correlation between WMC and effort $adj.R^2 = .16$, $N = 680$, $p < .001$). A 0.16 adjusted R^2 presented in figure 4.2 indicates 16% of effort can be explained by our predictor variable, WMC. According to the p-value (0.001) with confidence interval of 95%, this result is significant because it is lower than 0.5.

There was a strong positive correlation between WMC and SLOC $adj.R^2 = .79$, $N = 680$, $p < .001$). A 0.79 adjusted R^2 presented in figure 4.2 indicates 79% of SLOC can be explained by our predictor variable, WMC. According to the p-value (0.001) with confidence interval of 95%, this result is significant because it is lower than 0.5 .

Table 4.2: The adjusted R^2 of the variables in this study

	Effort	**SLOC**	**WMC**
Effort	1	0.21	0.16
SLOC	0.21	1	0.79
WMC	0.16	0.79	1

Step-wise linear regression

Tables 4.3 present the results of regression analysis. The first model (Model 0) uses the independent variable SLOC to model maintenance effort. SLOC does not explain maintenance effort ($adj.R^2 = .21$), which shows that SLOC is not enough to show differences in maintenance effort. When adding WMC to Model 0, Table 4.3 (Model 1) shows that WMC had zero effect on maintenance effort. The adjusted R^2 (0.21) shows that WMC and SLOC can account for 21% of the variation in maintenance effort.

Table 4.3: Results for regression analysis for effort.

	Model 0		**Model 1**	
	Adjusted R^2	Significance	Adjusted R^2	Significance
SLOC	0.21	0.001	0.21	0.001
WMC	NA	NA	0.000	0.72
Adjusted R^2	0.21		0.21	

4.2 Discussion

Hypothesis 1 - Correlation between SLOC and effort

The relatively low adjusted R^2 of 0.21 is reason enough to reject the hypothesis. There is no evidence of a strong linear correlation between total SLOC and effort metrics at a file level. The assumption that larger classes require more maintenance effort can be challenged. Other external factors influence maintenance effort more than SLOC. Factors such as experience and familiarity can affect maintenance effort more than SLOC.

Finding 3: SLOC and effort have a weak positive correlation ($adj.R^2 = .21$, $N = 680$, $p < .001$).

Hypothesis 2 - Correlation between WMC and effort

The relatively low adjusted R^2 of 0.16 is reason enough to reject the hypothesis. There is no evidence of a strong linear correlation between total CC (WMC) and effort metrics at a file level. The assumption that more complex classes require more maintenance effort can also be challenged. Other external factors influence maintenance effort more than total CC (WMC). Factors such as experience and familiarity can affect maintenance effort more than total CC (WMC).

Finding 4: WMC and effort have a low positive correlation ($adj.R^2 = .16$, $N = 680$, $p < .001$).

Answer to Research Question 2

Can we effectively use previously measured SLOC, CC, and effort to estimate/predict future maintenance effort?

The results from the empirical study suggests that neither SLOC nor MWC are enough to predict maintenance effort. The result from hypothesis 1 and 2 indicate the relationship between these variables are not strong enough. When predicting effort, many factors are omitted which may affect more the overall prediction. Factors such as experience and familiarity may play a big role. The assumption that bigger class or more complex class increases maintenance effort is not true. Many more class properties are needed before attempting to build such predictive model. The results from this thesis, alongside previous studies (Sjøberg et al. [52], Soh et al. [55]) suggest that prediction models based on a combination of previous effort and code smells could provide better results.

In the stepwise regression, there was no difference in Model 0 and Model 1. Using SLOC by itself or together with WMC result in the same adjusted R^2 of 0.21. WMC did not add any value to Model 0. WMC lack of added value could be a result of multicollinearity. According to Field and Miles [20], multicollinearity exists when there is a strong correlation between two or more predictors in a regression model. Table 4.2 shows a strong relationship between SLOC and WMC. Many other studies have also reported this strong correlation between SLOC and WMC [61, 15, 35]. However, when aggregating source code metrics such as CC over larger units of Java code, the dominating factor quickly becomes the number of units rather than the metric itself. Since the number of units is a factor of system size, aggregated CC indeed measures system size more than anything else [50].

Finding 5: SLOC and WMC are not enough to predict maintenance effort

Finding 6: SLOC $\sim$ SLOC + WMC. Using SLOC by itself or together with WMC produces the same adjusted R^2 of 0.21.

4.3 Threats to validity

This section discusses the threats to validity of our studies following common guidelines for empirical studies presented by Runeson and Höst [49]. It also provides preliminary recommendations.

Threats to construct validity - concern the design of our study. In our study, we used the data from the Sjøberg et al. [52]'s study. Developers were asked to perform their tasks in multiple rounds, which could lead them to learn. Learning bias can be an issue because we did not include it in our model.

Threats to conclusion validity - pertain to our correct use of mathematical tools. We used natural logarithm and Regression models to build our models. We use the implementation provided by python. Therefore, we believe that our results do not suffer from threats to their conclusion. However, future work should explore using other predictive models.

Threats to internal validity - concern our selection of systems, tools, and analysis method. LAT was validated for accuracy. The results from the accuracy assessment analysis suggest that we can be confident on the accuracy of the tool.. We have tested our tools that measure SLOC and CC but to mitigate any unknown issues we published our data and scripts.

Threats to reliability validity - concern the possibility of replicating this study. Every result obtained through empirical studies is threatened by potential bias from the used data sets[40]. To mitigate these threats, Sjøberg et al. [52] performed their study using six developers that developed and maintained independently four systems. Apart from that, it is possible to replicate this study, and the data[1] of this study is also available for replication.

Threats to external validity - concern the generalization of our findings. Because Sjøberg et al. [52] used six companies and four systems, we cannot claim that our results would apply to any software company or any systems with similar characteristics and context to those described in the study by Sjøberg et al. [52]. However, we are bringing new, interesting information regarding effort estimation.

[1]http://bit.ly/2wiyTIS

Chapter 5

Conclusions and future work

5.1 Conclusions

In this thesis, we analyzed the extent of which past effort, total CC and SLOC of classes can be used to predict future maintenance effort. First, we created a tool (LAT) that extracts effort from developers IDE activity logs. Then, LAT is validated empirically in terms of accuracy. Using the dataset of a previous study, we conducted an empirical analysis to examine the relationships between WMC, SLOC and maintenance effort. The results from our analysis suggests that:

- WMC has no strong linear correlation with effort
- SLOC has no strong linear correlation with effort

The results of this thesis suggests that neither SLOC nor MWC is enough to predict maintenance effort. So the assumption that bigger classes or more complex classes increases maintenance effort can be challenged.

5.2 Future work

In the future, it should be noted that more modern logging tool such as FLUORITE should be used to conduct new experiments. FLUORITE will enable better accuracy. LAT should be tested more extensively for accuracy with more data. Furthermore, replication of Sjøberg et al. [52] and Soh et al. [55] study should be done using LAT to confirm or refute their results.

Acknowledgement

Foremost, I would like to express my sincere gratitude to my supervisors Dr. A. Yamashita and Prof.dr. J.J. Vinju for the continuous support of my research assignment. Their guidance helped me in all the time of research and writing of this thesis. I could not have imagined having better mentors during the past few months. Furthermore, I want to thank my colleagues in the SWAT teams at CWI, especially my co-intern from the UvA, Adrian Zborowski for being such good company. Lastly, I would like to thank God and my family.

Thank you, UVA for this crazy, fruitful and unbelievable journey.

Bibliography

[1] Mohammad Alshayeb and Wei Li. An empirical validation of object-oriented metrics in two different iterative software processes. *IEEE Transactions on software engineering*, 29(11):1043–1049, 2003.

[2] Douglas G Altman and J Martin Bland. Measurement in medicine: the analysis of method comparison studies. *The statistician*, pages 307–317, 1983.

[3] Sven Amann, Sebastian Proksch, Sarah Nadi, and Mira Mezini. A study of visual studio usage in practice. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 124–134. IEEE, 2016.

[4] Jushan Bai and Serena Ng. Tests for skewness, kurtosis, and normality for time series data. *Journal of Business & Economic Statistics*, 23(1):49–60, 2005.

[5] Ayşe Bakır, Burak Turhan, and Ayşe Bener. A comparative study for estimating software development effort intervals. *Software Quality Journal*, 19(3):537–552, 2011.

[6] Hans Benestad, Bente Anda, and Erik Arisholm. Assessing software product maintainability based on class-level structural measures. *Product-Focused Software Process Improvement*, pages 94–111, 2006.

[7] L Breiman, JH Friedman, RA Olshen, and CJ Stone. Classification and regression trees. monterey, calif., usa: Wadsworth, 1984.

[8] UK Cambridge Dictionary. Cambridge dictionaries online, 2015.

[9] Sunita Chulani, Barry Boehm, and Bert Steece. Bayesian analysis of empirical software engineering cost models. *IEEE Transactions on Software Engineering*, 25(4):573–583, 1999.

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

[11] Samuel Daniel Conte, Hubert E Dunsmore, and Vincent Y Shen. *Software engineering metrics and models*. Benjamin-Cummings Publishing Co., Inc., 1986.

[12] Anna Corazza, Sergio Di Martino, Filomena Ferrucci, Carmine Gravino, Federica Sarro, and Emilia Mendes. How effective is tabu search to configure support vector regression for effort estimation? In *Proceedings of the 6th international conference on predictive models in software engineering*, page 4. ACM, 2010.

[13] Ignatios Deligiannis, Ioannis Stamelos, Lefteris Angelis, Manos Roumeliotis, and Martin Shepperd. A controlled experiment investigation of an object-oriented design heuristic for maintainability. *Journal of Systems and Software*, 72(2):129–143, 2004.

[14] T. DeMarco. *Controlling Software Projects: Management, Measurement & Estimation*. Yourdon Press Computing Series. Yourdon Press, 1982. ISBN 9780917072321. URL https://books.google.nl/books?id=y7AmAAAAMAAJ.

[15] Khaled El Emam, Saïda Benlarbi, Nishith Goel, and Shesh N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering*, 27(7):630–650, 2001.

[16] Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electronic notes in theoretical computer science*, 217:5–21, 2008.

[17] K Anders Ericsson and Herbert A Simon. Verbal reports as data. *Psychological review*, 87(3): 215, 1980.

[18] NE Fenton and SL Pfleeger. Software metrics: a rigorous and practical approach. 1997. *Brooks/Cole Pub Co*, 1993.

[19] Alan R Feuer and Edward B Fowlkes. Some results from an empirical study of computer software. In *Proceedings of the 4th international conference on Software engineering*, pages 351–355. IEEE Press, 1979.

[20] Andy Field and Jeremy Miles. Discovering statistics using sas. 2011.

[21] Beat Fluri. Assessing changeability by investigating the propagation of change types. In *Companion to the proceedings of the 29th International Conference on Software Engineering*, pages 97–98. IEEE Computer Society, 2007.

[22] Martin Fowler and Kent Beck. *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 1999.

[23] Davide Giavarina. Understanding bland altman analysis. *Biochemia medica: Biochemia medica*, 25(2):141–151, 2015.

[24] Juan Carlos Granja-Alvarez and Manuel José Barranco-García. A method for estimating maintenance cost in a software project: a case study. *Journal of Software Maintenance*, 9(3):161–175, 1997.

[25] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques.* Elsevier, 2011.

[26] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. In *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*, pages 30–39. IEEE, 2007.

[27] Israel Herraiz and Ahmed E Hassan. Beyond lines of code: Do we need more complexity metrics? *Making software: what really works, and why we believe it*, pages 125–141, 2010.

[28] ISO/IEC. *Software Engineering–Product Quality: Quality model*, volume 1. ISO/IEC, 2001.

[29] Anastasia Izmaylova, Paul Klint, Ashim Shahi, and Jurgen Vinju. M3: an open model for measuring code artifacts. *arXiv preprint arXiv:1312.1188*, 2013.

[30] Graylin Jay, Joanne E Hale, Randy K Smith, David P Hale, Nicholas A Kraft, and Charles Ward. Cyclomatic complexity and lines of code: Empirical evidence of a stable linear relationship. *JSEA*, 2(3):137–143, 2009.

[31] Magne Jørgensen. A review of studies on expert estimation of software development effort. *Journal of Systems and Software*, 70(1):37–60, 2004.

[32] Jacky Keung. Empirical evaluation of analogy-x for software cost estimation. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 294–296. ACM, 2008.

[33] Barbara A Kitchenham, Guilherme H Travassos, Anneliese Von Mayrhauser, Frank Niessink, Norman F Schneidewind, Janice Singer, Shingo Takada, Risto Vehvilainen, and Hongji Yang. Towards an ontology of software maintenance. *Journal of Software Maintenance*, 11(6):365–389, 1999.

[34] Paul Klint, Tijs Van Der Storm, and Jurgen Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *Source Code Analysis and Manipulation, 2009. SCAM'09. Ninth IEEE International Working Conference on*, pages 168–177. IEEE, 2009.

[35] Davy Landman, Alexander Serebrenik, and Jurgen Vinju. Empirical analysis of the relationship between cc and sloc in a large corpus of java methods. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 221–230. IEEE, 2014.

[36] Lucas M Layman, Laurie A Williams, and Robert St Amant. Mimec: intelligent user notification of faults in the eclipse ide. In *Proceedings of the 2008 international workshop on Cooperative and human aspects of software engineering*, pages 73–76. ACM, 2008.

[37] Hareton KN Leung. Estimating maintenance effort by analogy. *Empirical Software Engineering*, 7(2):157–175, 2002.

[38] Panagiotis Louridas. Static code analysis. *IEEE Software*, 23(4):58–61, 2006.

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

[40] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE transactions on software engineering*, 33(1):2–13, 2007.

[41] Tim Menzies, Ekrem Kocaguneli, Burak Turhan, Leandro Minku, and Fayola Peters. *Sharing data and models in software engineering.* Morgan Kaufmann, 2014.

[42] Roberto Minelli, Andrea Mocci, Michele Lanza, and Lorenzo Baracchi. Visualizing developer interactions. In *Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on*, pages 147–156. IEEE, 2014.

[43] Roberto Minelli, Andrea Mocci, and Michele Lanza. I know what you did last summer: an investigation of how developers spend their time. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, pages 25–35. IEEE Press, 2015.

[44] Gail C Murphy, Mik Kersten, and Leah Findlater. How are java software developers using the elipse ide? *IEEE software*, 23(4):76–83, 2006.

[45] Paul Oman and Jack Hagemeister. Metrics for assessing a software system's maintainability. In *Software Maintenance, 1992. Proceerdings., Conference on*, pages 337–344. IEEE, 1992.

[46] Martin Robillard, Robert Walker, and Thomas Zimmermann. Recommendation systems for software engineering. *IEEE software*, 27(4):80–86, 2010.

[47] Linda Rosenberg, Ted Hammer, and Jack Shaw. Software metrics and reliability. In *9th International Symposium on Software Reliability Engineering*, 1998.

[48] DJ Rumsey. How to interpret a correlation coefficient r. *Statistics For Dummies*, 2016.

[49] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14(2):131, 2009.

[50] Martin Shepperd and Darrel C Ince. A critique of three metrics. *Journal of Systems and Software*, 26(3):197–210, 1994.

[51] Martin Shepperd and Chris Schofield. Estimating software project effort using analogies. *IEEE Transactions on software engineering*, 23(11):736–743, 1997.

[52] Dag IK Sjøberg, Aiko Yamashita, Bente CD Anda, Audris Mockus, and Tore Dybå. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, 39 (8):1144–1156, 2013.

[53] Will Snipes, Emerson Murphy-Hill, Thomas Fritz, Mohsen Vakilian, Kostadin Damevski, Anil Nair, and David Shepherd. A practical guide to analyzing ide usage data. *The Art and Science of Analyzing Software Data*, 2015.

[54] Zephyrin Soh, Thomas Drioul, Pierre Antoine Rappe, Foutse Khomh, Yann Gael Gueheneuc, and Naji Habra. Noises in interaction traces data and their impact on previous research studies. In *Empirical Software Engineering and Measurement (ESEM), 2015 ACM IEEE International Symposium on*, pages 1–10. IEEE, 2015.

[55] Zéphyrin Soh, Aiko Yamashita, Foutse Khomh, and Yann-Gaël Guéhéneuc. Do code smells impact the effort of different maintenance programming activities? In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 393–402. IEEE, 2016.

[56] Ricardo Valerdi. 10.4. 2 convergence of expert opinion via the wideband delphi method: An application in cost estimation models. In *Incose International Symposium*, volume 21, pages 1246–1259. Wiley Online Library, 2011.

[57] Wil MP Van der Aalst and AJMM Weijters. Process mining: a research agenda. *Computers in industry*, 53(3):231–244, 2004.

[58] Joost Visser, Sylvan Rigal, Rob van der Leek, Pascal van Eck, and Gijs Wijnholds. *Building Maintainable Software, Java Edition: Ten Guidelines for Future-Proof Code.* " O'Reilly Media, Inc.", 2016.

[59] Karl E Wiegers. Lessons from software work effort metrics. *paper publicado no site www. processimpact. com, último acesso em jan*, 2002.

[60] YoungSeok Yoon and Brad A Myers. Capturing and analyzing low-level events from the code editor. In *Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools*, pages 25–30. ACM, 2011.

[61] Liguo Yu and Alok Mishra. An empirical study of lehman's law on software quality evolution. *Int. J. Software and Informatics*, 7(3):469–481, 2013.

[62] Rafdzah Zaki, Awang Bulgiba, Roshidi Ismail, and Noor Azina Ismail. Statistical methods used to test for agreement of medical instruments measuring continuous variables in method comparison studies: a systematic review. *PloS one*, 7(5):e37908, 2012.

[63] Horst Zuse. *A framework of software measurement.* Walter de Gruyter, 1998.