INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Departamento de Engenharia Eletrónica e de Telecomunicações e Computadores

A tool to simplify software log analysis

Nuno José Cancela Branco Pinheira Pereira

(Licenciado em Engenharia Informática e de Computadores)

Projecto para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientador : Doutor Artur Jorge Ferreira

Júri:

Presidente: Doutor José Manuel de Campos Lages Garcia Simão

Vogais: Doutor Nuno Miguel da Costa de Sousa Leite
Doutor Artur Jorge Ferreira

October, 2022

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Departamento de Engenharia Eletrónica e de Telecomunicações e Computadores

A tool to simplify software log analysis

Nuno José Cancela Branco Pinheira Pereira

(Licenciado em Engenharia Informática e de Computadores)

Projecto para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientador : Doutor Artur Jorge Ferreira

Júri:

Presidente: Doutor José Manuel de Campos Lages Garcia Simão

Vogais: Doutor Nuno Miguel da Costa de Sousa Leite
Doutor Artur Jorge Ferreira

October, 2022

To my family and friends

Acknowledgments

I would like to express my gratitude to my supervisor Prof. Artur Ferreira, for his help and guidance. I would also like to thank Prof. Matilde Pós-de-Mina Pato for the template used to write the dissertation.

Abstract

Log analysis is a necessary, challenging, and time-consuming task for software development and maintenance. As with many fields of Information Technologies, there is an effort in the optimization of the logging process as well as on the analysis of the stored information.

There is a severe lack of standardization on the log data representation, which poses challenges on the development of tools for log analysis. Another issue is the size of some log files, that can lead to slow searches.

In this work, we develop a standalone log analysis tool. This tool has an intuitive and simple interface, such that it has a small learning curve for the user. For a typical user, its use should be straightforward.

The open source tool provides some functionalities for text-based log files, displaying some indicators and graphs. Thus, it allows for the user to quickly locate the origin of the problems within the analysed code.

Another testing tool was developed in the context of this work. This tool is a configurable log file source generator providing the creation of different scenarios for log analysis.

Use cases were written for all features, being evaluated and tested, both manually and programmatically.

Keywords: Logging, Log analysis, Log parsing

Resumo

A análise de ficheiros de log é uma tarefa morosa e difícil mas necessária no desenvolvimento e manutenção de sistemas de software. Tal como muitas outras áreas das Tecnologias da Informação, existe um esforço por optimizar a forma como processamos e analisamos os dados contidos nos ficheiros de log.

Nota-se numa grave falta de standardização na forma como os dados presentes nos ficheiros de log são escritos. Isto leva a que o desenvolvimento de ferramentas para a análise de logs seja muito desafiante. Adicionalmente, a dimensão dos ficheiros de logs provoca dificuldades em termos de desempenho, em operações de leitura e procura.

Neste trabalho desenvolveu-se uma ferramenta para análise de ficheiros de log. A interface é simples e intuitiva, tendo como objectivo que a curva de aprendizagem para o utilizador seja o mais reduzida possível. Assume-se que o utilizador alvo tem conhecimentos básicos de programação.

Esta ferramenta será open source e fornece algumas funcionalidades para a análise de ficheiros de log, mostrando alguns indicadores e gráficos, permitindo ao utilizador localizar rapidamente a fonte dos problemas nos ficheiros analisados.

Adicionalmente, desenvolveu-se uma ferramenta para auxílio dos testes. Esta ferramenta é um gerador de logs configurável de forma a poder gerar diferentes cenários de teste.

As funcionalidades foram especificadas e foram validadas manualmente e por testes unitários.

Palavras-chave: Logging, Análise de logs, Parsing de logs

Contents

List of Figures

List of Tables

Acronyms

API Application Programming Interface. 12, 16

CSV Comma-Separated Values. 35

DIKW Data Information Knowledge Wisdom. 5

FAQ Frequently Asked Questions. 13

GUI Graphical User Interface. 20, 31

IO Input/Output. 12
IT Information Technology. 6

JDK Java Development Toolkit. 31
JRE Java Runtime Environment. 12
JSON JavaScript Object Notation. 35
JVM Java Virtual Machine. 45

LCP Longest Common Prefix. 29

NLPS Natural Language Processing System. 16

RFC Request for Comments. 15

SEC Simple Event Correlator. 16

XML Extensible Markup Language. 35

1

Introduction

1.1 The problem - log analysis

Log file analysis is a necessary and relevant part of software development, maintenance and correction. It is however a tedious and laborious process, in which many man-hours are spent analysing text files, due to the vast amount of data to handle. This process is a necessity since most complex bugs in software require the context and timeline that logs provide. Log monitoring is also a necessary task, both for maintaining the system's good functioning, for detecting attacks, and preventing failures.

With the ever-growing digitization trend of the world, new systems with their own challenges are constantly rolling out to production. Most software and digital equipment produce varied and extensive log files. These files may contain relevant information, which may be spread across the files which may vary in size from some Megabytes to many Gigabytes or more. The size of these files becomes a challenge in terms of storage and transmission, as well as increasing the effort needed to discern the information needed. An example of the information dispersion in a log file is shown in Figure 1.1, which is an excerpt of the run-time log from a tool developed for this dissertation. The purpose of this tool is to generate log files in a controlled manner to provide a way to accurately evaluate the solution proposed in this project.

As these tasks are important and mandatory, we should look for solutions and optimizations for them. There are solutions on the market to tackle these issues, however many developers perform these tasks manually, either because they are unaware of the

existing tools or due to the learning curve needed to master them. Additionally, many of these tools require a payed subscription.

The purpose of this dissertation, approached as a project, is to develop and evaluate a tool which facilitates the log analysis and the monitoring processes. The goal is on log parsing to help the user find patterns and search the logs for the necessary information. Additionally, metrics based on log data and metadata are developed and employed to provide simple real-time monitoring and organization features.

```
INFO: Invalid input {} expected Double
out 28, 2021 7:38:30 PM gui.screens.common.LabeledDoubleFieldComponent isCellValueValid
INFO: Invalid input {} expected Double
out 28, 2021 7:38:30 PM simulator.Simulator initializeSimulator
INFO: Initializing Simulator logger: {}, Filename: {}, Directory: {}
out 28, 2021 7:38:30 PM gui.screens.common.LabeledDoubleFieldComponent isCellValueValid
INFO: Invalid input {} expected Double
out 28, 2021 7:38:30 PM gui.screens.common.LabeledDoubleFieldComponent isCellValueValid
INFO: Invalid input {} expected Double
out 28, 2021 7:38:30 PM gui.screens.common.LabeledDoubleFieldComponent isCellValueValid
INFO: Invalid input {} expected Double
out 28, 2021 7:38:30 PM gui.screens.common.LabeledDoubleFieldComponent isCellValueValid
INFO: Invalid input {} expected Double
out 28, 2021 7:38:30 PM gui.screens.common.LabeledDoubleFieldComponent isCellValueValid
INFO: Invalid input {} expected Double
out 28, 2021 7:38:30 PM gui.screens.common.LabeledDoubleFieldComponent isCellValueValid
INFO: Invalid input {} expected Double
out 28, 2021 7:38:59 PM simulator.Simulator updateStatisticalModel
INFO: Updating Statistical Model
out 28, 2021 7:38:59 PM statistical.adhoc.AdHocStatisticalModel updateModel
INFO: Updating Model
out 28, 2021 7:39:03 PM gui.screens.common.LabeledDoubleFieldComponent isCellValueValid
INFO: Invalid input {} expected Double
out 28, 2021 7:39:03 PM simulator.Simulator updateStatisticalModel
INFO: Updating Statistical Model
out 28, 2021 7:39:03 PM statistical.adhoc.AdHocStatisticalModel updateModel
INFO: Updating Model
out 28, 2021 7:39:06 PM simulator.Simulator updateStatisticalModel
INFO: Updating Statistical Model
out 28, 2021 7:39:06 PM statistical.adhoc.AdHocStatisticalModel updateModel
INFO: Updating Model
out 28, 2021 7:39:12 PM gui.screens.common.LabeledDoubleFieldComponent isCellValueValid
INFO: Invalid input {} expected Double
out 28, 2021 7:39:12 PM simulator.Simulator updateStatisticalModel
INFO: Updating Statistical Model
out 28, 2021 7:39:12 PM statistical.adhoc.AdHocStatisticalModel updateModel
INFO: Updating Model
```

Figure 1.1: Example log file (a snippet of the file).

1.2 Objectives of the dissertation

The main goal of this dissertation is to develop a log analysis tool with the following key functionalities:

1. Display the contents of a log file and enable the user to search for errors, and keywords among other indicators and values.

2. Allow the user to filter the file's contents based on its elements and generate a new file with the results.

3. Compute and check the following metrics

    - File size.
    - Keyword histogram.
    - Keyword occurrences threshold.
    - Common word analysis.
    - Keywords over time.

4. Monitoring features

    - Based on the metrics, as defined above.
    - File growth over time.

5. Organization

    - Based on the keyword occurrences threshold, allow the user to move or copy the files into specific folders.

Keywords are portions of free text like 'Initializing' or 'An error has occurred' which can be searched for and matched within the source text. Associated with them are thresholds, these are either a set number of occurrences or the percentage of the occurrences in relation to the whole. As the file is analysed by the tool, the occurrences will be counted and matched to the thresholds defined, if the threshold is met or surpassed the threshold is triggered which then generates a warning in the tool.

Another goal of this work is to develop a log file source generator to properly evaluate the proposed log analysis tool. Figure 1.2 depicts this idea.

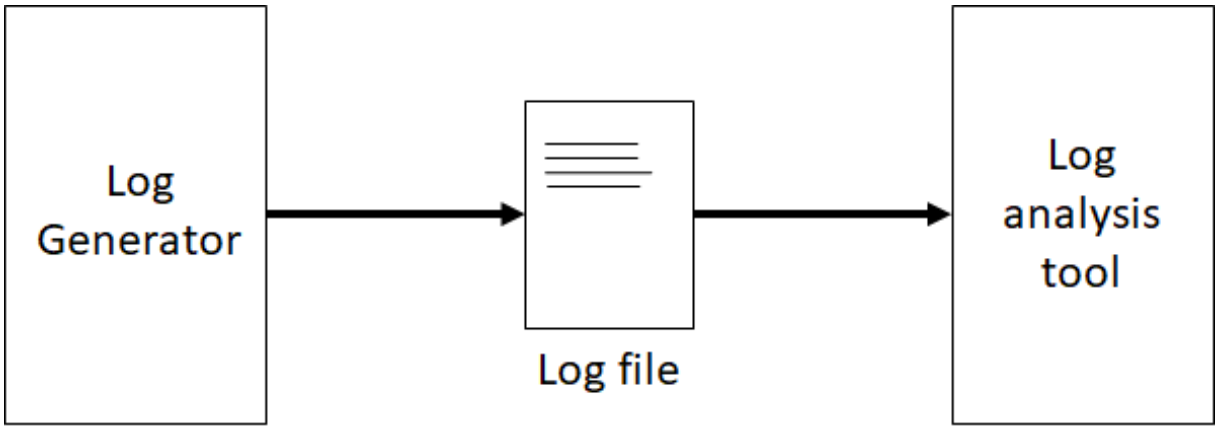


Figure 1.2: General tool interaction overview.

The log analysis tool is designed to analyse plain text log files. Additionally, each line of the text log file must contain an indication of time in the form of a Timestamp, a Date, or Time. It must have a definition of logging level as well as an user message.

1.3 Document organization

The remainder of this document is organized as follows.

Chapter 2 addresses the challenges about logging and the existing solutions for this problem and performs a review on the state-of-the-art.

Chapter 3 presents the proposed solution and its key features, including the log file source generator.

Chapter 4 covers the technical aspects of the implementation, on its key features about the development and the most important decisions taken in the development.

Chapter 5 reports the experimental evaluation carried out on the developed tools.

Chapter 6 contains the conclusions reached over the development of this dissertation as well as directions for future work.

On Appendix A, we have the requisites specification for the developed solution, in the form of use-cases.

On Appendix B, we present additional experimental results as an extension to those reported in Chapter 5.

2

State-of-the-Art

Log analysis is not a recent field of study. It has evolved along with the rest of the computer science field. This chapter provides an analysis of the developments made throughout the years on this matter. In Section 2.1, we discuss the need for logs, their origin, and history. In Section 2.2, we present the major challenges associated with log analysis. In Section 2.3, we address a taxonomy which represents the major components of logging and log files. Section 2.4 presents the Data Information Knowledge Wisdom (DIKW) model and its connection to logging. Some of the many logging frameworks are analysed and common traits discussed in Section 2.5. In Section 2.6, we cover some of the existing commercial tools for log analysis and monitoring. Section 2.7 reviews some of the existing algorithms to analyse log files. Section 2.8 briefly addresses the use of machine learning in this context. Finally, Section 2.9 gives an overview on the matters discussed in this chapter.

2.1 The need for logs

Logging is the act of keeping a log. This has been done throughout history and continues to be done in order to keep records of important events for historical reference, such as the log book on a ship, the average temperature on a city or the number of times a team wins a competition.

Outside of the computer science field, there are examples of the usefulness of logs, as seen on television when a detective visits a prisoner, or accesses the evidence locker,

a registry is kept. If something goes missing, or the prisoner ends up in trouble, the authorities can access those logs and trace the events. In computer science, logging is a tool which allows for the tracking of system events in a semi-persistent way. In software engineering, the principles are the same as for everyday life. When software is running in a production server and an issue occurs, there are often very few clues the users are able to give to solve it. Without the logging being carried out in the background, tracing the steps and finding the issue may be impossible, mainly if the situation at hand is not easily reproducible.

Log files provide an unbiased record of what is happening with the software. Even when there are no issues with the system, logging can be used to monitor the status of the system, the behavior of the users, and which parts of the product are more sought after than others [13]. Actively monitoring logs can help predict and prevent severe issues from happening, or at least reduce response time, helping to mitigate their effects. A common example is timeouts in method calls. If there is a monitoring tool, the issue can be swiftly investigated and a solution can be found (for example, restarting a lagging server). Logs can also be used to improve the system's overall construction and development. For example, if it is detected that there is an excessive amount of requests to a database, the offending code can be refactored, reducing the resource usage and increasing the systems performance.

Logging is of paramount importance in Information Technology (IT) systems' security. When developed properly, these logs record important events like password changes, improper accesses, malware detection, denial of service attacks, among others. From a security standpoint when these events are detected a red flag should be raised. For this to happen, the system's owners must monitor their logs. This can be done manually but for large scale systems an automated monitoring system can help reduce the workload and increase the effectiveness [42].

Additionally, companies are sometimes audited, and these audits rely not only on the data stored in physical form, or in the companies' databases. The auditing process can also rely on the system logs to evaluate any malpractice. The critical functionalities can be specially monitored and tracked for this specific purpose [51].

2.2 Challenges in log analysis

Logs are composed of semi-structured messages, which are created within the source code. Usually, they follow the same overall structure. The first few words contain the logging framework's structured text, including usually the timestamp, followed

by the variable text provided by the programmer. This portion of the message may follow a common structure within that system, containing fixed messages enriched by contextual information [1].

Modern systems have the tendency of combining software from different sources, some of which are open-source, some of them are in-house developed. Each subsystem has its own set of logging practices, and uses different frameworks among other factors [2].

The most basic approach to log analysis is based on simply reading the log files. A more advanced approach is based on the use of regular expressions and scripts to find known patterns in the text. Both of these approaches rely on the analyst's domain knowledge and are very time consuming [1, 2]. This approach presents other challenges, whenever there is a significant change to the software, the scripts may need to be reviewed. Additionally, domain knowledge can be easily lost with employee turnover.

With these challenges in mind, there has been a trend to apply machine learning techniques [8, 16] and automation to the log analysis and processing tasks [1].

2.3 Logging - a taxonomy

There are many types of logs as there are many types of systems. Some systems use several different logging strategies depending on their needs. Database systems usually keep records of the queries that were run, to allow analysts to verify or undo changes to a database or even restore it to a previous state in case of failure [31–33]. Another approach to logging is keeping record in a database of all events/changes that are considered critical. If a piece of software, or a given functionality has high importance, interactions with it must be logged in a database for persistent storage for auditing purposes. This is a similar concept to what is done when entering restricted areas in government facilities.

Instead of using a database, logs are often written into text files in the system's directory structure. The log messages may vary in format and source. For instance, they can be garbage collection logs, server logs, web logs, among others. They may have classifications according to their severity or importance. All of this is valuable information for maintenance, security and quality assurance of the software.

A taxonomy was defined [1] to categorize the different scopes of study related to logging, organized into four main topics:

1. Logging.

2. Log Compression.

3. Log Parsing.

4. Log Mining.

Figure 2.1 depicts these four topics and their connection. The log compression stage is optional.

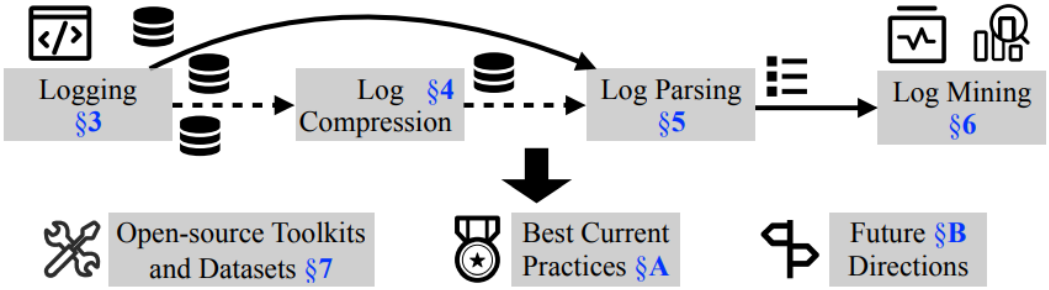


Figure 2.1: An overall framework for automated log analysis (adapted from [1]).

In the **Logging** aspect, the focus is on studying the ways programmers use the available frameworks to register information in the log files. The target is to improve the practices in logging namely, "where-to-log", "what-to-log", and "how-to-log". Where-to-log investigates the appropriate placement and amount of logging statements used to avoid over-use and under-use of logging, which will degrade the quality of the logged information. What-to-log targets the quality of logged information, including the logging level, static text, and dynamic text. How-to-log focuses on the best practices of logging in terms of design. Further research [4] has added the concept of "whether-to-log", to cover the dynamic adjustment of logging verbosity to match the run-time needs of the system.

Figure 2.2 shows these aspects of Logging.

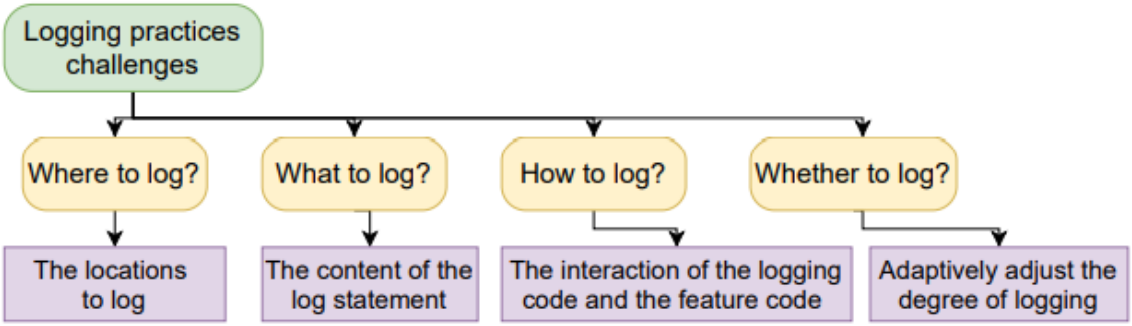


Figure 2.2: Logging practice challenges (adapted from [4]).

Log Compression, studies the challenges posed by the volume of data contained in the logs and the available tools to deal with them. As log files need to be stored and transmitted, they face the same challenges that images, audio and video, and other media files, as the cost of transmission and storage rises with the size of the files. One way to reduce the resulting files size, is by decreasing the verbosity levels which results in less messages but also in loss of information. Common file compression tools can help reduce the size of log files, but better results are achieved when the natural structure of log files is taken into consideration, namely the repetitive nature of static text.

Log Parsing, pertains to the processing of the existing log files in order to extract useful information. As log files aren't standardized, each having specific formats and patterns, they require specific approaches. The end result of parsing is to transform unstructured data into structured data such that a software tool can process. There are two modes of log parsing, offline and online. In offline parsing, all files are required to be available at the beginning of the process, and these logs are processed in batch. When the model is considered out-of-date it must be manually updated. Online parsing processes the files in stream, making this approach useful for pipeline processing.

Log Mining, covers the challenges and techniques used to automate log analysis. The used techniques, are based on machine learning [8, 16], data mining [54], and statistical models to automate the analysis of log files. Its purpose is to analyse large volumes of logging data to find patterns and trends. These techniques are then used for software maintenance and failure prediction, among other purposes.

As pointed out in [4], there is no standard way to refer to the log messages. Some refer to them as "log entries" or "log records", and distinguish the fixed part as a header and the variable part as the message. We will opt to refer to each line in the log as a log message, and its sub-components as header and body, given that we consider that it makes it clearer for the reader.

2.4 The Data, Information, Knowledge, and Wisdom model

The Data, Information, Knowledge, and Wisdom model (DIKW) is illustrated in Figure 2.3. It is a four level model used in the study of information theory. As presented in [5], there is some disagreement within the scientific community in what these levels represent, and what enables the transitions between them. Nevertheless, we find the model useful to systematize the different levels of analysis over data. As there is disagreement within the community, several models and interpretations of this hierarchy have been proposed and discussed. One of the models, adapted from our interpretation, is presented in this section.

Data is at the bottom of the pyramid. By adding context to the data we discover Information. Information when associated with a meaning or structure becomes Knowledge. With insight taken from Knowledge we obtain Wisdom. Data is described as individual facts, digits, signals, among others, such as a color, or a number. Information is Data within a context, it is defined as structured Data; for instance the color we see is of the sky and the number is on a thermometer. We attain Knowledge by giving meaning to Information. As an example, the sky is blue indicates that it is a clear sky. The number on the thermometer states that it is cold. Finally, Wisdom comes from the insight derived from the Knowledge. For instance, the sky is blue and clear, so it is less likely that it will rain soon. The temperature is very low, so you need to dress accordingly.

Applying the DIKW model to logging, the raw logs are at the level of Data. As pointed out in [4], the log parsers transform unstructured raw log files into sequences of structured events. The task of the parser can be interpreted as transforming Data into Information. This process is what enables the subsequent use of automatic techniques to interpret events. By applying machine learning models and artificial intelligence techniques, the information can be driven further into knowledge. As a consequence, it reduces the effort from the users in gaining knowledge from the original data.

As described before, log files are at the level of data, and effort is required in order to change that data into information. Changing the information into knowledge requires additional effort. The same goes for obtaining wisdom from the information. Understanding these definitions and the transitions between layers helps all actors involved in the system's maintenance and development to take into consideration the effort needed. Knowing the effort required can inform decisions, such as investing in tools which decrease effort, or the automation of some processing.

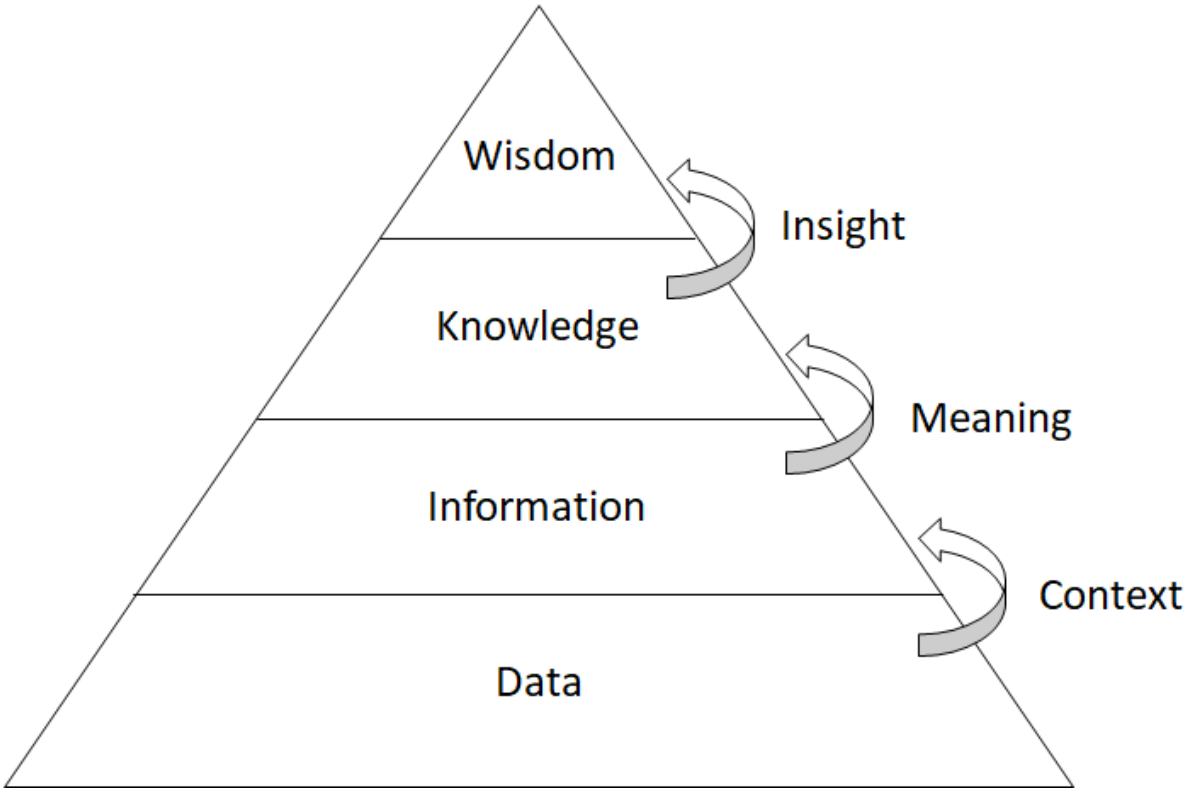


Figure 2.3: The DIKW model (adapted from [5]).

2.5 General characteristics of logging frameworks

Currently, there is a myriad of programming languages, and every year new programming languages and frameworks appear and disappear. Looking into all available languages is not the purpose of this dissertation. The focus is on the most popular programming languages of the last decades. The chosen languages are C/C++, Java, Python, and C# [50]. In the following subsections, for each programming language an example of the logging output is presented as well as the logging levels. Additionally, a small overview of the framework's relevant features is provided.

Logging levels help indicate and manage the severity and verbosity of a log message. In a well developed system, the most common log messages are informative, denoting the major steps and events taken by the system. There are levels of priority and relevance to the messages. Critical issues are usually rare, but carry a high level of importance, for example an Exception thrown by a method, or an error code 500 in a web-server. These issues demand attention and usually correspond to a severe failure in the system. Below this severity level, are the warning messages, which are usually more common. These messages can report minor issues in the system, but they can

also report impending failures. A torrent of warning level messages may predict an upcoming system failure.

Within normal production setups, the most common messages are of information level. These relate the normal flow within the code as well as relevant values that were exchanged or recorded, such as the access of a new user to the system, by recording the corresponding username. Lastly, there are debug level messages, which are usually not enabled in production systems, unless there is troubleshooting going on. The debug level logging is highly verbose and can significantly increase the size of the log files, due to the fact that debug level logging traces exhaustively the flow within the code and the data transferred by the system.

Ideally, all systems would have the most verbose levels always on, but this has a negative impact on performance from the Input/Output (IO) access to the files. There is also the issue of occupying disk space to store log files that may never be used (for a given time-window as old log files can be deleted). Both of these issues make this approach unappealing [38].

2.5.1 C/C++

Glog [17] is a library developed by Google for logging in C++. It offers customization to the user.

The example below was taken from the code itself [18]. The default setting contains the level information as the first character, the date and time, the thread identifier, file name and line number.

```
I1103 11:57:31.739403 24395 google.cc:2342] Process id 24395
```

The log levels are the following: FATAL, ERROR, WARNING, and INFO [19].

2.5.2 Java

Java Logging Application Programming Interface (API) is a framework. It comes with the core Java Runtime Environment (JRE), introduced in Java 1.4. In this framework, the simplest of the default settings that generates text is the "SimpleFormatter". This formatter generates two lines for each log entry, as shown below.

```
oct 13, 2021 7:53:37 PM (...) isCellValueValid
INFO: Invalid input {} expected Double
```

The log levels are the following: SEVERE (being the highest value), WARNING, INFO, CONFIG, FINE, FINER, and FINEST (lowest value).

Log4J framework is an Apache Licensed API. The example below is taken from the documentation on the configuration of the library [27].

```
17:13:01.540 [main] ERROR MyApp - Didn't do it.
```

The log levels are the following: OFF (to turn off logging), FATAL, ERROR, WARN, INFO, DEBUG, and TRACE [27].

2.5.3 Python

Logging [39] is the Python built in framework. The example log message was extracted from an official tutorial [40]. As can be observed in the cited tutorial, the Logging framework from Python has a very high level of customization. Some of the simplest examples of log messages don't even contain a timestamp.

```
2010-12-12 11:41:42,612 is when this event was logged.
```

The log levels are the following: CRITICAL, ERROR, WARNING, INFO, and DEBUG.

2.5.4 C#

Serilog [43] is a logging framework for .NET and the following message is adapted from their site.

```
09:14:22 [Information] Processed { Lat: 25, Lon: 134 } in 034 ms.
```

The log levels are the following: FATAL, ERROR, WARNING, INFORMATION, DEBUG, and VERBOSE.

Log4Net [28] is an Apache Licensed API framework developed for the Microsoft .NET runtime.

The example below can be found on the Log4Net Frequently Asked Questions (FAQ) page [29]. The first value on the example is a number which indicates how many milliseconds have elapsed since the start of the execution. This is as opposed to the one used in Serilog, for example.

```
176 [main] INFO  examples.Sort - Populating an array of 2 elements
```

The log levels are the following: OFF (to turn off logging), FATAL, ERROR, WARN, INFO, DEBUG, and ALL.

2.5.5 Common aspects of logging frameworks

In general, logging frameworks have some common traits in their outputs. There is a log level or severity of the message. There is a time indication (with different degrees of detail) and there is the message payload or body which carries the user defined message and code context. The code context is the indication of the class and function where the log message has been written.

Each logging framework however produces its messages with different formats, different order of elements and some even produce more than one line per message. This shows of the lack of standardization for logging tools, and increases the challenge of developing a generic tool to analyse any type of textual logs.

The logging levels as seen above are not the same in each framework. There is some overlap, but there are variations in number and semantics. When the frameworks are used correctly, the order of importance of the logging levels is inversely proportional to their occurrence in the log files. This is due to two factors, firstly the higher importance logs should be scarce as they point out failures, errors or exceptional behaviors. The second factor is that lower importance log levels such as DEBUG are generously distributed throughout the code, in order to maximize the available information. These are considered of lower importance as each line of log produced has very little information, as compared to a single critical log message.

Regarding the date/time format, there is also a high degree of variance as well. Some frameworks don't use timestamps in their messages as a default setting. Those who do, print the considered date and time in different formats, having no standardization for the date-time formats.

2.5.6 Syslog - a possible standard

Syslog [49] was created by Eric Allman and is a standard for consolidating log messages from different origins and using different formats. It pre-processes the received messages, by adding some information, normalizing the format and structure into a common format. Although there are several similarities to the frameworks described

above, such as a severity identifier, time register elements, among others, the standardization of all logging frameworks is not the focus of Syslog. The standard's scope is transforming messages from different sources and converting them into a format, which can be easily sent through the network into a logging server acting as an aggregation point for the information. The definitions made in the Request for Comments (RFC) [49] provide an interesting structure which could be followed as an universal standard, which is lacking. However, the purpose of the standard is not to define guidelines for all logging frameworks.

2.6 Existing tools

There are many tools in the market which help to manage logs in a more efficient manner. From the existing products, four of the most popular are briefly reviewed in this section. We covered these as they seemed to be the most relevant at the moment in the market. Our comparison on their features and capabilities cannot be adequately reduced into a tabular view, so we present their features individually, as each tool has its merits.

DataDog [12] allows for searching, filtering and analysis of logs without the need of a complex query language. It provides trend discovery, correlation of data and the definition of alerts. Allows for the unification of logs from different sources and helps with the scaling of log volumes. The cost varies from 15$ to 23$ per month, per user.

Fluentd [15] allows for the unification of data collection. It provides searching, filtering, and alerting. It has a vast library of community developed plugins which allow the user to customize the tool for their specific needs. It is open-source.

Logstash [34] converts events from multiple sources and creates a common log format for ease of use and analysis. It is open-source.

Loggly [30] allows for system performance and behavior analysis as well as the monitoring of key resources and metrics. It provides issue tracing and issue correlation identification. Prices vary from 79$ to 279$ per month, for an unlimited number of users.

The common trend on all these services is the facilitation of searching, filtering and analysing log files, which seem to be the core features. Each product adds its own features that help differentiate them from one another.

All of these tools are useful in their own way, serving different purposes. The purpose of the tool developed in this project is to provide the core functionalities found in the

analysed tools, but to deliver them in a simpler way. The features may not be as diverse or as powerful, but there is a need for a tool which is simple, and easy to use, being better than a standard text editor.

2.7 Algorithms for log analysis

There are many algorithms which, although they are not specifically designed for log analysis, they can be adapted for that purpose. In this section, we review some of these algorithms.

Regex, also known as regular expressions is a non-standardized system which allows the user to define patterns for text matching. There is no standard for the system, though it is widely used and supported by the major programming languages. In general, the regex system works by a set of rules which are specific for the platform in which they will be used. Each of these rules is defined by a character or a set of characters. Each set of characters is then combined to create a specific semantic. For example, based on the java regex Pattern [23] "\s" indicates a white space, and "[abc]" indicates a single character which may be "a", "b", or "c". In practice, this may be used for simple operations, such as splitting a phrase by its white spaces:

```
String[] split = line.split("\\s+");
```

Using the code above, if the variable "line" contains: "File search algorithms" then the result will be the array ["File", "search", "algorithms"].

Regex can also be used to search for highly complex patterns by aggregating any number of logical sets. A slightly more complex example to search for dates in the YYYY-MM-DD or YYYY-M-D format is: "\d{4}-\d{1,2}-\d{1,2}".

Simple Event Correlator (SEC) [52] is an open-source tool developed by Risto Vaarandi for detecting and correlating events. It is lightweight and platform independent. It uses a rule-based approach for detecting and correlating events. It is written in Perl, which enables it to run in a wide range of operating systems and takes advantage of Regex for searching files. It can be used as a standalone application or it can be used as part of another system through an API.

Grok is an algorithm developed by Klaus K. Obermeier, to analyse technical documents. In his paper [37], in which the algorithm is presented, its application was for the study of medical descriptive texts for patients with liver diseases. The paper proposes a system to analyse natural language also known as a Natural Language Processing

System (NLPS). Natural language processing faces challenges, as computers are not able to interpret language the same way as human beings. Text can vary wildly when the domain it refers to changes as well as when the language it is written in changes. A person who is able to read two different languages can easily read a text and its translation. A computer system is unable to do so easily. What Obermeier proposed is the development of a system with a specific target domain, narrowing its scope. The system, when configured, cannot easily be adapted to another domain. Having this as a presupposition, the algorithm is able to search and identify different parts of written text and to classify them according to their semantics.

2.8 Machine learning techniques for log analysis

Artificial intelligence [41] and machine learning [8] techniques are being applied to log analysis and processing. From the existing algorithms, the two most commonly found in the literature are Classification and Clustering [3].

Classification [3] operates by applying predetermined class labels to the targets of the classification, in this case, files. It requires a setup in which the class labels are defined and the classifier is trained. There is a training set in which each file has a class label assigned to it. Based on this training set, the classifier creates an internal classification structure, such as a decision tree for example. This technique can be used to quickly sort through large amounts of files, classifying each one by their relevance or priority. This algorithm can also be applied to classify individual messages.

Clustering [3] can be used to group files based on their shared characteristics and similarities. These algorithms work based on the similarities, and the result of this process are groups of files which are considered similar within the scope of the algorithm's parameterization. This can be used for organizing large amounts of files into logical groups. If we have a set of logs from different origins, we may need to group them based on that characteristic for ease of analysis. Clustering can also be used within a file; in this case, the similarities are computed between messages. If a message belongs to a group interpreted as an irregular behavior, then it's likely that it is a message tied to an irregular behavior as well, and it is flagged as such.

2.9 State-of-the-art overview

Logging and the tasks associated with it are an unavoidable part of modern systems. We face many challenges dealing with this necessity, such as the volume and variety of data, and combining different sources into the same file. To better help us understand the existing wealth of information and knowledge, a taxonomy was created by several researchers, creating a common language and organizing the relevant topics. This helps us navigate this vast field, as well as improving the communication over these topics.

From our research into a subset of the Logging field of investigation, we determined that although logging frameworks share many characteristics, they follow no standard. Additionally, we analysed some of the existing tools which help users tackle log analysis and system monitoring, as well as an overview of algorithms and machine learning techniques which can be applied to the logging field of investigation.

3

Proposed Solution

In this chapter, we will describe the proposed solution. In Section 3.1, we cover the basic outline of the solution while in Section 3.2 we present the key features of the application. In Section 3.3, we describe the parsing component of the solution. Section 3.4 covers the monitoring metrics in detail. In Section 3.5, we address the log analysis feature. Section 3.6 covers the log generation tool. In Section 3.7, we discuss text search algorithms in light detail and which one we chose to use.

3.1 Solution outline

From the research made into logging as a field which we present in Chapter 2, we will focus on log analysis, which is a small portion of this field, but with direct implications in everyday work for many developers. Although there are many available tools, they can be far too complex for most developers' needs. On the other hand, tools like Notepad [53] software are useful, but they don't provide tailored features. What we intend to do is to develop a tool within that spectrum, closer to the less complex side.

We will focus on the log parsing topic of the logging taxonomy. As we need to be able to parse the log files, so we can turn data into usable information. With this transformation, we will then provide the user with tools for log analysis.

In this chapter, we present the functional and non-functional requisites for the application. The requisites we will present are supported by the use cases and supplementary specification found in Appendix A.

As stated in Chapters 1 and 2, we intend to develop a tool which provides support to log analysis. The tool is a desktop application, which will require minimal installation, as we want the user to have as little hassle as possible when using and setting up the tool.

The tool should be able to be used in any modern operating system. People in charge of maintaining and developing systems may work in vastly different operating systems and technological stacks. So, we find it appropriate to not lock out users by creating a platform specific tool.

The tool will not have users or login requirements, it is meant to be used as any integrated developing environment or text editing software. There is no online, cloud or remote server needs. The whole tool runs locally and uses the resources available in the machine in which it was launched.

In terms of user experience and responsiveness there are some requirements. We are aware that log files may vary wildly in size, so we define ranges of sizes and expected response times for parsing. Files under 10 MB should not exceed 5 seconds of response time. Files between 10 MB and 100 MB should not take more than 60 seconds. For the remaining possibilities, we do not find it reasonable to define requisites for response time. In that category, typical text editing tools will have trouble opening the files, in part due to the strain on the resources of the machine. The response time is linked to both the severity of the issue and the subjective perception of the user, so the defined time may be acceptable for some situations, and unreasonable in others.

Figure 3.1 represents a macro view of the tool, with two types of file inputs: log files and configuration files. Both types of files may be edited directly by the user, though it is not recommended for the configuration files.

The tool produces a Graphical User Interface (GUI) representation of the analysis and metrics, as well as new log files which result from the user's filtering. The application's GUI text is displayed in English language.

In Appendix A, we present the full specification document with all the functionalities described in a high level analysis.

3.1.1 Link between the DIKW model and the tool

As mentioned in Section 2.4, the DIKW model is a useful way to visualize the different stages which transform data into wisdom. The focus for this software is to take the data contained in the log files, and transform it into information. This transition is

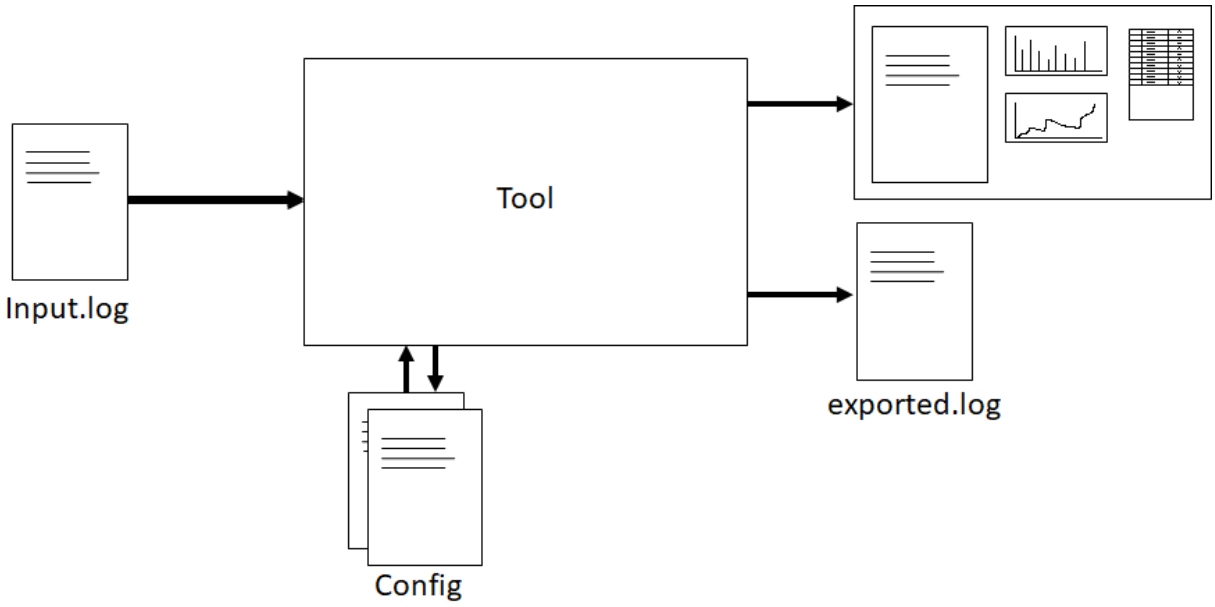


Figure 3.1: Macro level diagram of the proposed solution

meant to support the analysis process, reducing the effort. The remaining steps which take information to higher levels fall on the user. Figure 3.2 shows, side-by-side, the relationship between the DIKW model and our work.

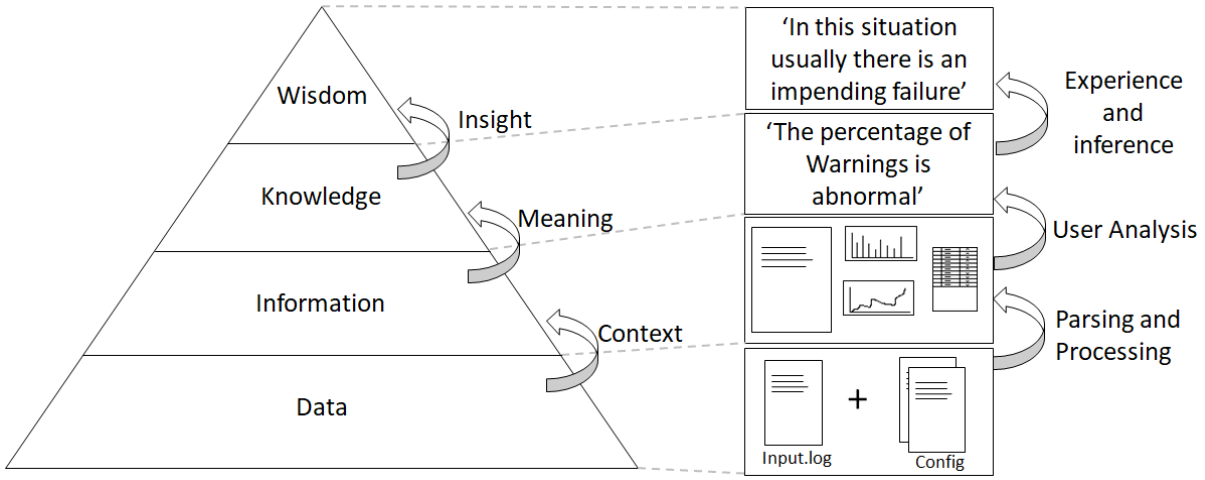


Figure 3.2: Comparative analysis between the DIKW model and the application

3.2 Features overview

There are two major features in this tool: log analysis and log monitoring.

Log analysis is based on the supposition that the user needs to read through log files

and analyse them by hand. This feature and its sub-features are targeted towards reducing the user's workload. The analysis performed in this feature is over a static file, and the metrics used in the visualization are meant to give an organized view over the file.

Log monitoring is focused on giving the user the capability of monitoring a file at runtime, discerning information and getting notification of events. The metrics applied in this case are targeted to catch events in real time. For example, if we define a warning to be launched when an exception appears in the log, we can quickly notice that there is an ongoing issue.

The metrics we are targeting in this work have a small scope to avoid scope creep. Even so, the tool is developed in a way which should make the addition of new metrics a relatively simple matter.

3.3 Parsing

As described in Section 2.5, there is no standardized way of generating log messages. This creates a real challenge to develop a general purpose tool. There are some expectations on the way the logs will present themselves but there is no sure way of knowing a log file's actual structure until there is contact with it. Even so, as logging frameworks are highly customizable, even within the same framework, the default expectations can be biased or wrong.

From our research, we determined that there are several components to a log message. There is usually a static portion created by the framework and a free-text portion defined by the user [1]. These elements together form a text string, the individual components which form the log message are usually separated by a character. We will refer to these portions of the message as "Text Classes" from now on. The ones identified as actionable are:

- Timestamp - An aggregation of date, time and other data such as a timezone.
- Date - A date in some format.
- Time - The time in some format.
- Identifier - An identifier for a thread, process or another element.
- Origin - The class (a Java class for example) or an equivalent construct where the log message originates from. Additionally it may be, or contain, the method or function or an equivalent element such that it was the source of the log message.

- Level - The log level.
- Developer Message - The variable content message written by the developer.

Log files may contain other text which does not follow the general structure described above. Exceptions and other events can overrule the common structure per line or lines, as shown in Figure 3.3. These are considered as part of the previous log message.

```
Exception in thread "main" java.lang.NullPointerException
        at gui.common.customComponents.profile.ProfilePanel.createComponents(ProfilePanel.java:41)
        at gui.common.customComponents.profile.ProfilePanel.<init>(ProfilePanel.java:33)
        at gui.occMonitor.KewordOccMonitorPanel.createProfilePanel(KewordOccMonitorPanel.java:67)
        at gui.occMonitor.KewordOccMonitorPanel.createComponents(KewordOccMonitorPanel.java:30)
        at gui.occMonitor.AbstractOccMonitorPanel.<init>(AbstractOccMonitorPanel.java:37)
        at gui.occMonitor.KewordOccMonitorPanel.<init>(KewordOccMonitorPanel.java:21)
        at gui.MainScreen.createTabbedPane(MainScreen.java:59)
        at gui.MainScreen.createComponents(MainScreen.java:51)
        at gui.MainScreen.<init>(MainScreen.java:32)
        at EntryPoint.main(EntryPoint.java:9)
```

Figure 3.3: Exception log example

When writing the logging statements, the developers may also add information to the "Developer Message" portion, which will disrupt the expected message structure. For example, in a data access layer the user may log a query. If the query is complex, it may be logged as a multi-line message, which breaks the expectation of one line per message.

As there is no standard in which we can rely on, there are two obvious options, either create a very narrow scope for a specific logging framework, or create a recognition system to detect what logging style is being used. However, neither of these options suits our needs, the first creates a tool with a very narrow focus, and the second option requires a highly complex recognition system. The conclusion to which we arrived is that the tool is meant for users who already know how to analyse a log file, so we should leverage the knowledge from the user.

The approach taken is to give the user the capability of configuring parsing profiles which are tailored for the system under analysis. This requires that the user knows how to extract the general structure of the log file, but that is a reasonable expectation. We rely on the user to define the general structure of the message, indicating how one expects it to be parsed.

3.3.1 Parsing profiles

We will rely on the information given by the user to configure the parsing system. For that purpose, we created parsing profiles.

A parsing profile is a way for the user to tell the tool how the log files should be parsed. The user defines the components of the message and the symbol that splits them. The profiles are then stored in a persistent way, allowing the user to define them once and reuse them as needed.

The use cases which cover this feature are described in Appendix A, on Sections A.1.8, A.1.9, A.1.10, and A.1.11.

Log messages are divided into different elements, such as Log Level or Origin. In this context, we will refer to those elements as Text Classes, and the elements which split them are Separators. The Text Classes are the ones identified in Section 3.3.

The Separators are:

- " " - White-space.
- "\t" - Tab.
- "-" - Hyphen.
- ":" - Colon.
- ";" - Semi-colon.
- "," - Comma.
- "." - Full-stop.
- "[" - Open bracket.
- "]" - Close bracket.

The user may want to ignore portions of the log messages, thus the parsing profile definition will allow the user to ignore any of the elements. By analysing the structure of the log messages, we found that sometimes the character used to split some text classes appears several times, so we also allow the user to define how many occurrences of that separator symbol as needed.

An example:

```
2021-01-01 06:41:04.000 INFO Test Message
```

In this example, we can see that there are four components. First a date, followed by time, then we have a logging level indicator and finally the message. In this case, we

would need to define either a date and time classes, separated by a space character, or a timestamp, which includes both date and time. In that situation however, we would need to skip the first white space, or the timestamp would only include the date.

Due to the myriad of possible configurations of the log messages, we define pre-requisites for the tool to be able to parse them. Each line must contain an indication of time in the form of a Timestamp, a Date, or Time. It must have a definition of logging level as well as an user message. Additionally, profiles must have a distinct name.

3.4 Monitoring metrics

The monitoring metrics we propose to implement are:

- File size.
- Keyword histogram.
- Keyword occurrences threshold.
- Common word analysis.
- Keywords over time.

A keyword histogram displays the occurrence of keywords in proportion to one-another. This will allow the user to visually discern the volumes of the chosen keywords, enabling quick decisions.

Both a threshold and a specific number of occurrences are viable and valuable ways to look at text occurrences. If the user wishes to guarantee that as soon as a certain keyword or phrase appears, there is a warning triggered, then the number of occurrences as an indicator is the best option. On the other hand, if the user expects a certain level of messages to appear but an increase is problematic, then a threshold is better suited.

For the metrics section, connector words will be filtered out to avoid having high rates of irrelevant words such as "the". This task is known as stop word removal [35]. This will not be done in the analysis part, so as to make sure that the text is readable by the user.

The use cases which cover this feature are A.1.7, A.1.12, A.1.13, A.1.15, and A.1.16.

In order to persist the configurations for the metrics, we use metrics profiles which follow a similar logic to the parsing profiles. These will was well be written into configuration files, registering the configurations made by the user.

3.4.1 Keywords

The keywords definition became necessary as we needed a way for the user to link a set of characters (keyword) to a specific state. This state may vary, as we provide various options. The user defines the keywords in the metrics profile editor, and they are stored within the metrics profiles configuration files. There are the following options:

- Keyword - this is the text that is meant to be monitored and searched.
- Case sensitive - this configuration indicates to the tool if the keyword text is meant to be taken exactly or if the characters are meant to be interpreted ignoring the case.
- Threshold type - this indicates if the user desires to define a threshold at all, or if so, if it is when the value is equal to, above, below, etc.
- Threshold unit - this further enriches the threshold, it is meant to be considered the number of occurrences, or the percentage of those occurrences.
- Threshold value - in this part the user indicates what is the number of occurrences (when the unit is occurrences) or the relative percentage of occurrences (when the unit is percentage).
- Warning level - this was a later addition, allowing the user to tie a level of importance to the event generated when the threshold definition is met.

3.5 Log analysis

The log analysis feature will start by presenting an overview of the file's content, based on metrics such as log level distribution, and most common words. From this overview screen, the user can quickly decide if further analysis will be of value.

If the user decides to further analyse the file, a new screen is available. This screen allows the user to visualize the log in a structured way. It also allows the user to filter and search the log. Additionally, the user can export portions of the log into new log files. Extracting a portion of a log file based on a time frame, for example, is also useful, given that there is a way to identify the start and end of the time-frame. Another useful approach is filtering it by log level or by the origin of the message.

The use cases which cover this feature are A.1.1, A.1.2, A.1.3, A.1.4, and A.1.5.

3.6 Log generation tool

In order to test the accuracy of the metrics, we aim to develop another tool. This tool will generate mock log files with configurable ratios of log levels, with a randomization element. The tool will also be based on existing examples from different technologies to test the parsing and confidence levels of the metrics among different frameworks.

As show in Figure 3.4, the tool will receive inputs from two sources, the user and references. The user will indicate what is the desired distribution of log levels, as well as the target style to be used.

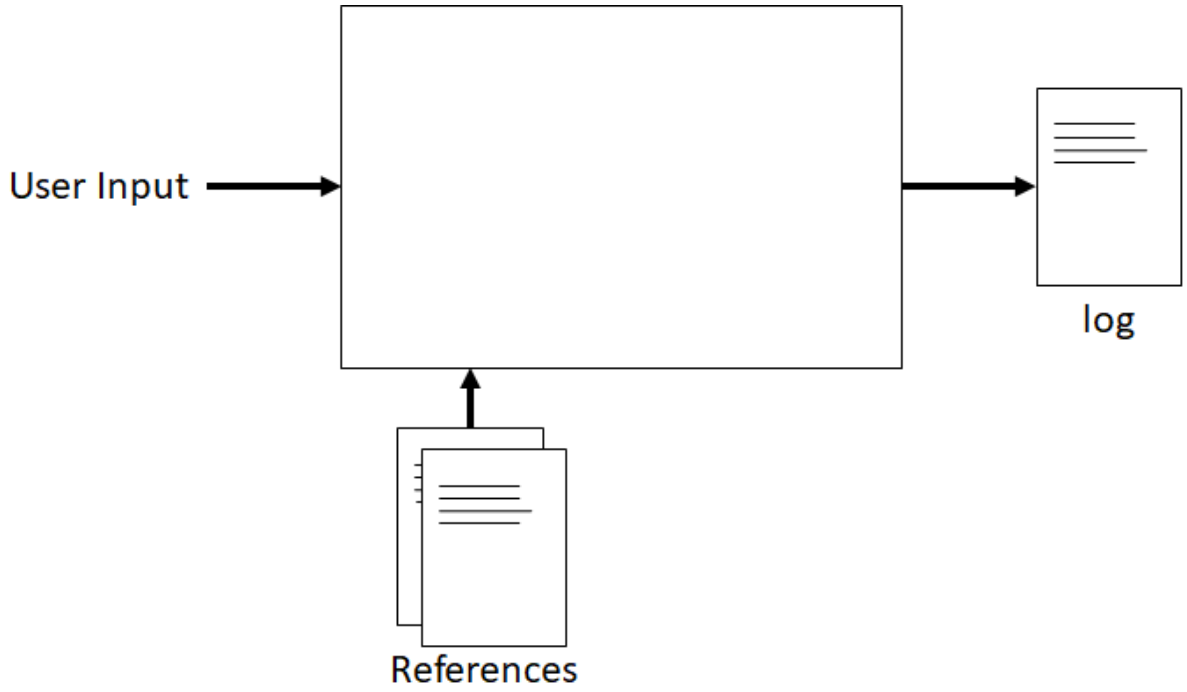


Figure 3.4: Macro level diagram for the log generation tool

We opted not to use real log files as the information contained within may contain business logic and sensitive information. Additionally, with real files we cannot control the distribution of the information and create results whose validity is easy to confirm.

3.7 Text search algorithms and data structures

As with any file, log files may reach large sizes, if we want to have searching capabilities, we need to analyse through the contents of the file. In a naive approach, we may run through the file character by character, comparing them to our target value. This

is not an efficient way to analyse large volumes of text. To tackle this difficulty, many string searching algorithms have been proposed. We present some of these algorithms which we found relevant, though there are others.

A **Suffix Tree** [46] is a tree composed by all the suffixes of a source text. Each path taken in the tree is a valid suffix. As such, the tree occupies much more space than the original source, as many more combinations need to be stored, at least in the simplest implementation of the algorithm.

Suffix trees can serve various purposes, such as in information technologies, to facilitate searching. They have other applications as well, such as in Computational Biology, where they can, for example, be applied to analyze DNA protein sequences [45].

We will now explain the simplest implementation of the algorithm. The source text is traversed starting from the end. In the first iteration of the algorithm, the window encompasses one character. For each following iteration, the window size increases by one. If our source text is CAGTCAGG then for the first iteration the window would capture the last character of the string which is "G", in the next iteration it would be "GG" and so on.

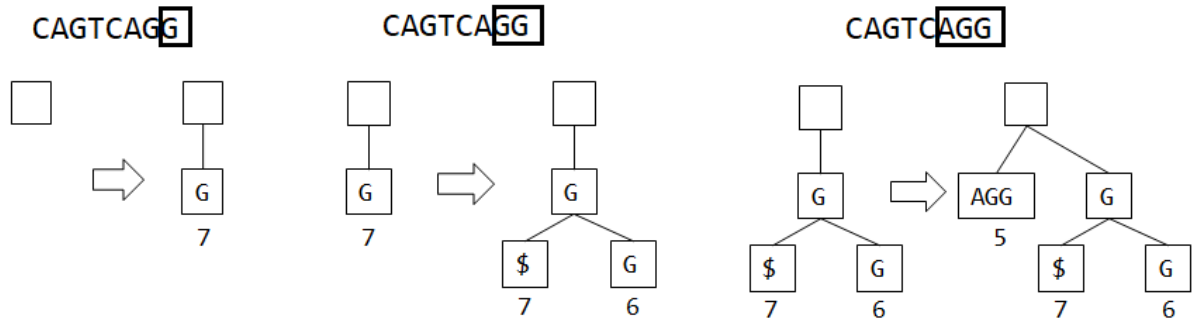


Figure 3.5: Suffix tree construction: first iterations example.

In each iteration, we take the first character of the window and compare it to the existing nodes of the suffix tree head. If there are no nodes with the corresponding character we create a new one, with the full contents of the window. If there is a match, we drill down into that node, checking the child nodes for matches to the window, in this case however, we discount the match we found previously. For our example text, in the first iteration there are no nodes, so we simply add a new node with "G" and store the current index on the source text, in the second iteration there is a match, the first character in the window matches the first node, so we move down into that node. It does not have any child nodes, so we create a new node "G" under it and store the current index on the source text, as well as a "$" node, which is an empty node meant to store the position of the original "G" node. Figure 3.5 exemplifies the first three iterations.

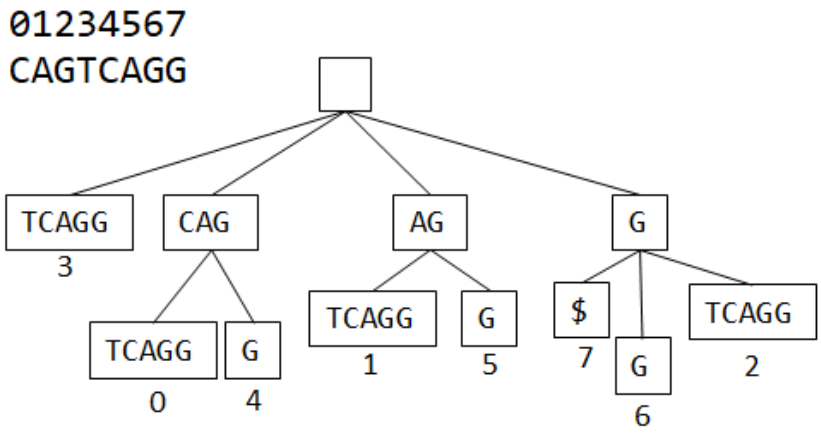


Figure 3.6: Complete suffix tree.

This process is repeated for the whole source text creating a tree. For our example text, the resulting tree is shown in Figure 3.6.

The search process is quite similar to the building process, however, instead of adding nodes we search the tree until the text is fully matched. After finding the last node for the match, we have the sub-tree which contains all positions for matches for the source string.

For more information on this algorithm, please see [46–48] and the references therein.

The **Suffix Array** [14], is based on the same ideas as the Suffix Tree. The algorithm for construction of the Suffix Array starts by generating an array with all suffixes and the position of each one in the source string. The array is then ordered in lexicographical order. From this point forward the suffixes can be discarded, this allows the Suffix Array to have much smaller space requirements when compared to Suffix Trees.

Another structure which is useful in the algorithm is the Longest Common Prefix (LCP). This support structure registers the number of equal characters from the start of each prefix. This is calculated after the sorting of the suffixes. Through the use of this structure, we can reduce the amount of string comparisons, by comparing the size of the string we wish to find, and the number of common characters.

For more information on this algorithm, please see [14] and the references therein.

Boyer-Moore [10] works by placing the pattern to find P against the source text T lined up at the start and then comparing P to T backwards, stopping as soon as there is a different character. The text to find is then shifted to a new position and compared again against T. The shift calculation is based on two algorithms. Bad Character and Good Suffix. The Bad Character algorithm compares the miss-matched character in the source and attempts to match it to another position in the P. If it finds it, P is aligned

with that match, if not, P is fully shifted by its length, as there will be no match between the bad character in T and the characters in P. A simplified explanation of the Good Suffix rule is that when a Bad Character is found, the successfully matched characters so far are searched as a pattern within P to the left of the bad character, if there is a match, then P is shifted to the right so it lines up with this new found match and the comparison restarts from the rightmost character of P.

For more information on this algorithm, please see [10] and the references therein.

Aho-Corasick [6] works by creating a graph from a set of words we desire to find. The advantage of this algorithm is that the graph may be stored to be reused saving the time needed to recreate it. It is mainly useful for searching for several patterns at once. For our purposes, as we are focused on searching for a single value at a time, this is not a useful approach. It has a linear cost as it has to match the generated graph to the full text.

For more information on this algorithm, please see [7] and the references therein.

3.7.1 Application

Regarding the usage of suffix trees for the developed tool, each individual string is relatively small, with both a low time in terms of creating the suffix tree itself, but it also allows us to minimize wasted space and effort. With very large documents, many branches in the suffix tree would never be used, as we need to create branches and nodes for all possible combinations. This is a downside, as there is waste in terms of effort and space. Additionally, as we are dividing the original document into smaller, atomic portions, we can parallelize the process, reducing the time needed to process the entire set substantially.

In a similar way, the application of suffix arrays can be parallelized as each message is processed individually. Achieving the same gains, but with a lower memory cost, and a slightly slower search speed.

A limitation of using data structures like suffix trees or suffix arrays is that if we want to allow for non case-sensitive search we would need to create the structures with that in mind, as the latter in terms of programming is represented with different characters if it is upper-case or lower-case. This would constitute an additional overhead in terms of time, but more importantly in terms of space, effectively duplicating the amount of space required.

4

Solution Development and Implementation

In this chapter, we will describe the technical aspects of the implementation of the proposed solution. We start by presenting the chosen technical framework in Section 4.1. In Section 4.2, we describe the general architecture of the solution. Section 4.3 covers the approach used to develop the software. In Section 4.4, we discuss the technical decisions made when dealing with parsing and metrics profiles. Section 4.5 covers the development of one major feature, the log file analysis. Section 4.6 covers the development of the log file monitoring feature. In Section 4.7, we present an overview of the organization functionality of the tool. Section 4.8 presents the overarching technical decisions made for the development of the tool.

4.1 Technical framework

This project was developed using the Java framework. The Java Development Toolkit (JDK) is the OpenJDK Runtime Environment Corretto-11, from Amazon [9]. These language and framework were chosen due to the familiarity with it as well as the cross platform support for JRE based software.

The GUI was developed using the Swing framework native to Java. This framework is time tested and still commonly used for desktop software.

The software was developed with a modular approach in mind to allow for easier expansion and maintenance.

4.2 Architecture

The system architecture is composed of three layers as shown in Figure 4.1. The Presentation layer contains both the Views and the Presenters, the classes within are organized in packages according to their function. The Domain layer contains the Entities and the Services, grouped based on their function. The Data layer only contains a single subsystem for Data Access. As this system does not require a database, being based on files, the data layer is reduced in size and scope.

These layers were chosen based on the "Three-tier architecture" [20], separating the components by their purpose.

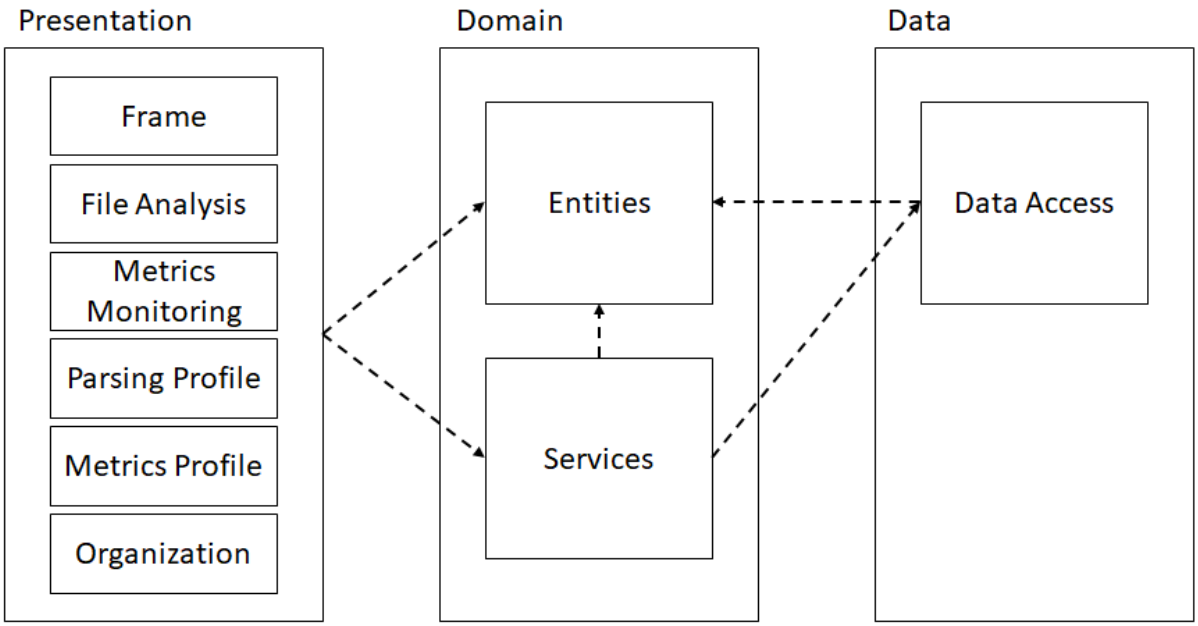


Figure 4.1: General architecture diagram

The **Presentation layer** is the frontier between the system and the user. Its purpose is to collect and present information to the user, and to communicate with the internal services. This layer has two main components, the views and the presenters. As a rule of thumb, for each view there is a presenter.

The views' purpose is to interact with the user and to exchange data. The purpose of the presenters is to manage the flow of information and execution of the view. The

presenter also communicates with the Domain layer, bridging the Presentation layer and the Data layer.

Each subsystem in the Presentation layer represents a set of views and presenters organized with their functional purpose in mind. For example, all the components found in the File Analysis subsystem only pertain to that functionality.

The **Domain layer** contains two subsystems, the Entities and the Services. The Entities subsystem has three subdivisions resulting in different packages. This division is meant to separate the domain entities by function, which makes using them easier.

The Services subsystem has only a single package, as each service contains within itself all the features that the Presentation layer will need. We opted not to have a service per presenter to reduce the amount of classes, reducing the system's complexity. The Services are responsible for bridging the Presentation layer and the Data layer. They are also responsible for implementing the business logic.

The **Data layer**, as this system does not use a database nor a database system, instead we use configuration files, the choice is justified in Subsection 4.8.3. The classic Create, Read, Update, and Delete (CRUD) operations are implemented but they do not require isolation levels. The concurrent access is controlled by keeping single access to the files.

4.3 Development approach

In Appendix A, we show the use cases which describe with a high level of abstraction the features and behavior of the solution to implement. From the use cases, we created a list of features to develop. Within each feature there are sub-tasks to further reduce the complexity of each task. The features were internally organized into GUI, business, and data tasks. Each task should be self contained and, if there are dependencies to other tasks, these dependencies should be minimized by using mocked behavior. If the dependencies cannot be circumvented, then the task is postponed until the dependency is solved.

The development process started with the GUI. This was chosen as the process of developing it would give us information on the soundness of our mock-ups and on the limitations of the technology. This proved to be an advantageous decision, as the presuppositions made in the initial schematics were not accurate. This approach allows us to develop the screens with a focus on the utility of the screens and not on the simplest way to match the screens to the domain entities.

The initial task, which was not derived from a specific use-case, was to create a skeleton

for the application. The classes and packages defined in the architecture were created, as well as the base JFrame, the "Look and Feel" of the application as well as two classes with constants. The "Look and Feel" of the application is the set of graphical assets that the Swing framework will use to construct the GUI. There are several options for this configuration [22], the one that was chosen is the "SystemLookAndFeel" which uses the assets that match the system where the software is running.

The two classes with constants are GuiConstants and GuiMessages described as follows. GuiConstants contains all the strings used in titles, buttons, and labels as well as the preferred dimensions for the windows and fields. The GuiMessages class contains all the messages used in the GUI. There are several advantages to doing this, for example, there is no need to repeat strings throughout the code. If there is a need to change the contents of a message or a label, you only need to edit the code in one place. Additionally, this would allow for an easier switch in the presentation language.

In terms of the development technique, we follow a Feature Driven methodology. Each use case defined in Appendix A is broken down into major features, moreover each major feature is broken down into minor features. Each major feature is only be considered complete when all of its minor features are completed. Each minor feature is only be considered finished when its requirements are developed and covered with unit tests. There will be no unit testing for GUI components.

This approach allows for the segmentation of the whole project into manageable portions. It also allows the reduction in complexity of each feature. Additionally, as each minor or major feature is implemented, a tangible artifact is produced.

Whenever improvements to the software are proposed or discovered, a new task will be created to keep track of them. Additionally, refactoring of code will be performed whenever a significant improvement can be achieved in relation to the time required to perform it.

4.4 Profiles

We chose not to use a database in this tool to keep it light and portable. There are lightweight databases like SQLite [44] which generates small database files, do not require installation or a server. A database like this would be a viable solution, but we opted to minimize the usage of third party software. We explain the reasons for that in Subsections 4.8.2 and 4.8.3.

Instead of a database, we use text files to store the persistent information. There are

several standardized formats like JavaScript Object Notation (JSON) [26], Comma-Separated Values (CSV) [11], or Extensible Markup Language (XML) [55], but none of them suits our needs. JSON is very popular as it is easily read and understood by both human beings and machines. We considered using this format but decided against it as we would need to either use a third party parsing library or to implement our own. CSV is not as easily read by human beings but is still manageable with some effort. The way CSV stores its values does not work well with the data we want to store so we opted not to use it. But it has the advantage of not needing a third party library. XML has a steep learning curve, and is the least accessible to be read as-is. If we used XML, we would also use a schema file to guarantee the consistency of the generated files. XML is too cumbersome for our needs. It would also require a third party library to process.

In general, we opted to use third party software as little as possible. We consider that the possible gains do not compensate the risk of using a third party library which we do not control and could introduce vulnerabilities into our software. We expand on the issue on Subsection 4.8.2.

In Section 4.4.1, we present the parsing profiles, which enable the user to configure a parsing strategy. In Section 4.4.2, we present metrics profiles, which are used to configure the metrics to be used in monitoring and analysis, as well as enabling or disabling those metrics.

4.4.1 Parsing profiles

We created our own format for storing information relating to parsing profiles. It is as follows:

```
Parsing Profile
Name;<Profile Name>
START-PROFILE
<Text Class>;IGNORE
<Text Class>;KEEP
<Text Class>;KEEP-SPECIFIC;<Specific Format>
<Separator>;SKIP;<Number of skips>
END-PROFILE
```

The text between the "<>" is chosen by the user. The <Text Class> is one of the available text classes and the <Separator> is one of the separators, <Number of skips> indicates

to the algorithm how many occurrences of the <Separator> character will be ignored when determining the limits of a <Text Class>. Between the "START-PROFILE" and the "END-PROFILE" lines, you can add any number of lines, with the caveat that the first line should be a text class and that between text classes there must be a separator. If the user wants to have more than one profile in the same text file, then simply add a new "Name" line after the "END-PROFILE" line and add a new structure as explained above. An example:

```
Parsing Profile
Name;Parsing2
START-PROFILE
TIMESTAMP;KEEP
SPACE;SKIP;1
LEVEL;IGNORE
HIFEN;SKIP;0
MESSAGE;KEEP-SPECIFIC;zd{4}-zd{1,2}-zd{1,2}
END-PROFILE
Name;Parsing3
START-PROFILE
TIMESTAMP;KEEP-SPECIFIC;yyyy-MM-dd HH:mm:ss.SSS
SPACE;SKIP;1
LEVEL;IGNORE
HIFEN;SKIP;0
MESSAGE;KEEP
END-PROFILE
```

These two profiles represent approaches into parsing messages. The first one 'Parsing2' states that it expects a timestamp which is to be kept, this can be achieved by skipping a space, after the timestamp there is another space, then the logging level, which is to be ignored. Following this, there is an hyphen, which separates the previous element from the next one, which is a message. In this case, the user decided to include a regex in order to format the message as it is read. The second profile is 'Parsing3'. This one also has a timestamp which is kept, but the user provides a specific format, which the timestamp should follow, here we also skip the first space in order to get the whole timestamp. Then, the level is ignored and the message is kept.

4.4.1.1 Testing parsing profiles

When developing the parsing profiles GUI we came upon an issue. As we create the profile, we had no way of validating it without launching an analysis of a file. To avoid the waste of time we added a new functionality out of necessity, and decided that it would be useful for the end user as well. In Figure 4.2 we provide an example of its usage, gaining immediate feedback for the profile we are creating. In the window where the user creates a parsing profile, there is another tab. In this tab, the user can test a string by inserting it in the text box and pressing 'test'. The tool will then apply the parsing profile, the user is currently creating, to the text and show what the parsed fields are.

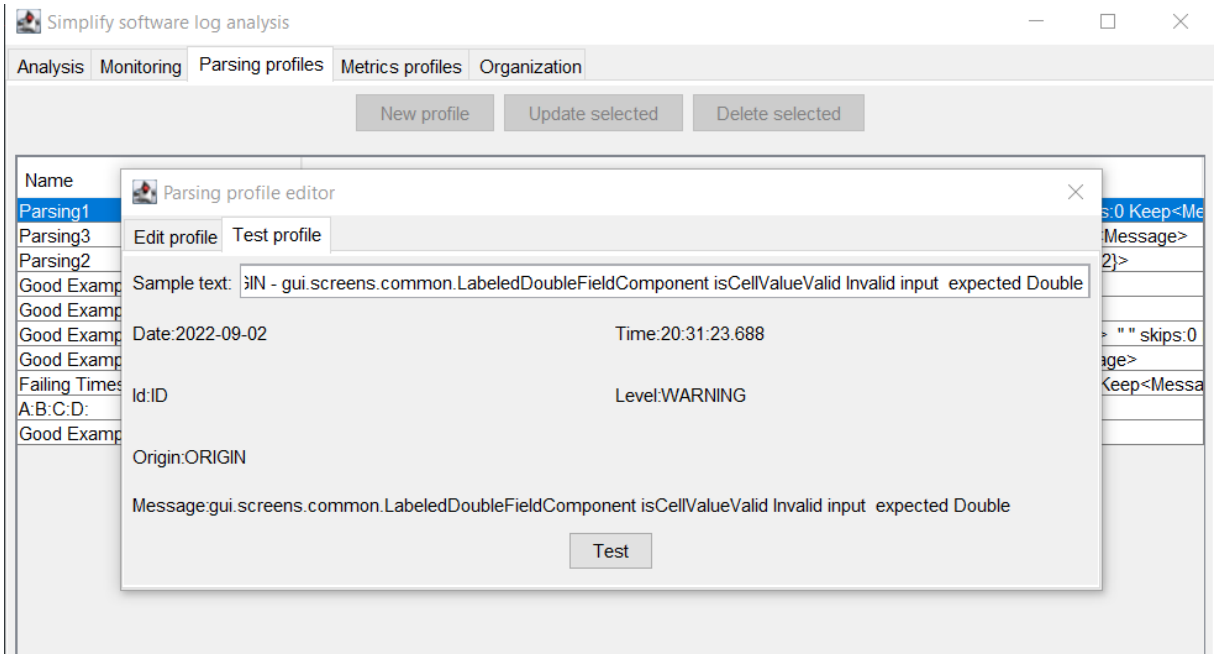


Figure 4.2: Parsing profile validation example.

4.4.2 Metrics profiles

For the metrics profiles, we created a system which follows the same logic of the parsing profiles:

```
Metrics Profile
Name;<Profile Name>
START-PROFILE
{MOST-COMMON-WORDS-TOKEN};<true/false>
{FILE-SIZE-TOKEN};<true/false>
{KEYWORD-HISTOGRAM-TOKEN};<true/false>
```

```
{KEYWORD-OVER-TIME-TOKEN};<true/false>
{KEYWORD-THRESHOLD-TOKEN};<true/false>
START-KEYWORDS
<Keyword Text>;<true/false>;<Threshold Type>;<Numerical Value>;
    <Threshold Unit>;<Warning Level>
END-KEYWORDS
END-PROFILE
```

The text between the "<>" is chosen by the user, within the constraints defined in the supplementary specification, Section A.2. As described in Section 4.4.1 the "START-PROFILE" and "END-PROFILE" tokens set the beginning and end of a profile definition. There are five static tokens, which enable or disable the types of metrics which will be employed. Within the "START-KEYWORDS" and "END-KEYWORDS" tokens you can have any number of keyword definitions. An example of a file with two profiles is as follows. The <Keyword Text> defines the string which will be measured, <true/false> indicates if the text is case sensitive or not, <Threshold Type> defines the type of threshold check, if it is above a certain value, below among others. The <Numerical Value> indicates the threshold value, <Threshold Unit> defines if the value is the number of occurrences or a percentage of the total. Lastly <Warning Level> defines the level of warning associated with the keyword threshold.

```
Metrics Profile
Name;Test1
START-PROFILE
MOST-COMMON-WORDS-TOKEN;true
FILE-SIZE-TOKEN;true
KEYWORD-HISTOGRAM-TOKEN;true
KEYWORD-OVER_TIME-TOKEN;true
KEYWORD-THRESHOLD-TOKEN;true
START-KEYWORDS
Kws1;false;NOT_APPLICABLE;0;NONE;CRITICAL
Kws2;true;NOT_APPLICABLE;0;NONE;HIGH
Problem;false;EQUAL_TO;1;OCCURRENCES;NONE
message;true;BIGGER_THAN;10;OCCURRENCES;MEDIUM
END-KEYWORDS
END-PROFILE
```

The profile above is a metrics profile. The five lines after "START-PROFILE" indicate that the metrics are enabled. Then, between "START-KEYWORDS" and "END-KEYWORDS" there are four keyword definitions, the first value is the text for the keyword, followed by a boolean which indicates if the keyword is case sensitive. Then, the type of threshold followed by the number or percentage for the threshold. After that, the indication if it is a number of occurrences or a percentage. Lastly, there is an indication of the severity associated to the threshold being met or surpassed.

4.4.3 Reading and writing

For writing profiles and to generate the format described above, we created the class ParsingProfileFunction and the MetricsProfileFunction which implement the Function interface. These interfaces receive an entity which represents the profile. For the parsing profiles it is an instance of the ParsingProfile class, and for the metrics profiles, an instance of the MetricsProfile class. From these entities, the functions generate a String object with the textual representation described earlier in Sections 4.4.1 and 4.4.2. The writer classes (ParsingProfileWriter and MetricsProfileWriter) have an instance of the respective function classes. This allows the writers to only have to concern themselves with opening/creating a file and writing to it. If for some reason there were two instances trying to concurrently write into the same file, the Java framework already deals with this not allowing concurrent writes into a file. We are using a BufferedWriter which is synchronized and thread safe.

In a similar way, the recovery of the profiles from the text files is done by a generic reader class. There is a ParsingProfileConsumer class and a MetricsProfileConsumer class which implement the functional interface Consumer. These classes are prepared to input line by line, the contents of the file and, using a simple state machine, process the text into one or more profile objects.

There was no need of implementing serialization [24] as we are not writing the actual objects and their state into the files. We are creating a representation of the configurations made by the user, and not the actual Java object itself.

This approach has two main objectives. First, to reduce the local complexity of the classes, helping to keep the single responsibility principle. The second objective is to reduce the effort needed if someone wants to add a new type of format. To do this, one would mostly need to create a new Function and a new Consumer with the same signature.

4.4.4 Profile management

As profiles, both parsing and metrics, may be written into several different files there was a need to manage them at run-time. The ParsingProfilesMemoryRepository and MetricsProfilesMemoryRepository classes were created to manage the profiles. The classes follow the Singleton design pattern, to guarantee that there is no multiplication of conflicting data structures. These classes are a centralized point for managing the profiles. In each class, we organize the existing profiles, grouping them by their origin file. They are also able to determine if a profile already exists without the need to read the files again. This, of course, relies on a reasonable usage of the software, if the user decides to edit profiles manually after the tool is launched, they do so at their own risk.

At startup, the tool loads all the existing profiles in the default parsing profile folder "ParsingProfiles" and from the metrics profiles folder "MetricsProfiles". All profiles are loaded into their respective memory repositories, a map is kept in which each origin file is mapped to the profiles contained within. Whenever a new profile is created, the repository validates if a profile with the same name and origin already exists. If so, the creation fails and is reported as such to the user. If not, the profile is created into the default profiles file. When the user decides to edit an existing profile, the memory repository is used to determine the existence of the profile, as well as to determine which profiles are contained within that same file. A shortcoming of this approach, is that whenever a profile is deleted or updated we need to rewrite the whole file. We see this as a minor issue. There is no realistic scenario in which this limitation would be a realistic impediment, as the profiles are manually changed through the GUI.

4.5 Log file analysis

In order to enable the user to remove unnecessary words from the occurrences counting, we added a line in the configuration file which contains some common words which are not likely to be relevant for the analysis. The configuration file uses the same format as defined before, a token which identifies the line and the contents for that configuration. The default line is the following.

```
STOP_WORDS;a;that;the;and;at
```

The user may edit this configuration, adding or removing words. For the new configuration to be read, the user must launch the application again.

Additionally, we want the user to be able to see, at a glance, if there are any relevant warnings present. For this purpose, we added another configuration to the file.

```
WARNING_COLORS;NONE;FFFDF7;CRITICAL;C21510;
    HIGH;DE4909;MEDIUM;DE9009;LOW;E6CC0E;INFO;17ABE6
```

This configuration maps the possible warning levels defined in the metrics profiles to a color scheme. There were two options for the color scheme, one would allow the user to define the desired color for each keyword and threshold. This would allow the maximum level of customization. The second option is to define a single set of colors, mapping each warning level to a color. We opted for the second option. Although having a high level of customization can be an asset, it would severely increase the work needed to create keyword configurations. It would also require the user to remember which colors had been picked for other profiles if standardization was desirable.

4.6 Log file monitoring

As we had developed the log file analysis first, along with the infrastructure to support it, the bulk of the work had already been done. We added a dynamic reader in order to keep monitoring the file while it is updated, with a boolean which allows the user to stop the process at any time.

The parsing component was reused, however, the metrics processing needed a different approach, for each new line we had to generate an updated report on the metrics and update warnings. We initially opted not to group the messages read but instead update the GUI as each message arrived, providing a more immediate feedback at the cost of more updates of the view. However the refresh rate of the GUI was causing severe latency in the actual presentation of the data. We moved to another approach, having a wait period accumulating messages and then updating the GUI in bulk.

4.7 Organization

The organization feature is another use that we can give to the parsing and metrics functionalities. The code for reading the files and profiles, as well as parsing and applying the metrics to the information gathered is already done for the remaining major features. For this feature, we analyse the set of files chosen by the user, those which the

thresholds defined in the metrics are surpassed, are either moved or copied to the target folder. This is easily done through the already existing file interaction framework in the JVM.

4.8 Overarching technical decisions

As referred in Subsection 4.4.3, we used Consumers and Functions in order to decouple the actual reading and writing process from the interpretation process. This was done also for the log file readers and exporters. An additional advantage of this approach is the ease of testing. By decoupling the interpretation process from the reading and writing process, we remove the need to use actual text files in order to test the behavior. By emulating the inputs for the classes, we can test exactly what we expect to get in the end.

Using the same approach, we added the use of a configuration file, which is stored in the file system.

The code is available at https://github.com/NunoJP/Tool/tree/master.

4.8.1 Tool startup

The tool is deployed as a JAR file. The user needs to have Java 11 JRE, or other compatible versions, running on the system in order to use this software. A batch file has been included to simplify the launch of the tool in a Windows environment. As this is a Java application, the tool can be run in most operating systems as long as they support the JRE. Upon startup, the tool creates a set of folders which store the configuration file as well as the parsing and metrics files. It also produces a configuration file with the default values, this file is not overwritten in subsequent startups. If the file is deleted, a new file will be created upon the next startup. The tool does not require installation. However, there is a limitation, if the tool is run in an environment in which it cannot create folders, then it will not launch.

4.8.2 The use of third party software

As mentioned in Section 4.2 and Section 4.4 we opted to minimize the usage of third party software, by using only what is not feasible to implement within our time constraints. We made this choice in order to reduce exposure to other software licenses, as we intend to allow this code to be open-source. This choice was also made with the

intent of reducing the exposure to software vulnerabilities present in code which we do not control.

4.8.3 Not using a database

Databases open possibilities to an application, you can query data, keep history, and in the case of relational databases, add structure to information. All of these capabilities are useful in many other applications, but for our purposes they are not needed. After parsing the information into memory we use the algorithms mentioned in Section 3.7, having no need to store the information into a database expecting to decrease search times.

In Section 5.3, we present the analysis of the searching capabilities of some of the algorithms. We conclude that searching without the support of a database achieves acceptable time results.

We also do not keep a history from previous sessions as the tool works in a real-time analysis scenario. We don't have users or need to store user related information. For these reasons, the added weight of using a database is not justifiable.

5

Experimental Evaluation

In this chapter, we will present the experimental evaluation of the developed tool. We start by stating the settings used for the evaluations in Section 5.1. In Section 5.2, we demonstrate the log generation tool. Section 5.3 covers the tests and results obtained while testing the search algorithms. In Section 5.4, we present the analysis functionalities of the developed tool. Section 5.5 covers the analysis for functionalities of the log file analysis. Section 5.6 presents the file operations features associated with the file analysis. In Section 5.7, we present the functioning of the organization functionality of the tool. Section 5.8 describes the results obtained.

5.1 Evaluation settings

The tests we present in this chapter were made in one computer, with the following specifications: CPU with 2.6 GHz, 12 GB of RAM, SSD hard drive with 380 GB free. No programs were running except for the background processes and the IDE.

Each section of each test was run at least five times, the first run took more time than the following runs for all scenarios as there is optimization done by the Java Virtual Machine (JVM) for subsequent runs.

5.2 Log generation tool

A log generation tool was developed in order to enable us to test our application. We present here two tests made with the tool, we used two different configurations for different log levels. The log files were then analysed using Notepad++[36] a commonly used tool.

The first configuration is shown in Figure 5.1. It has 4 levels. We generated a file with 5000 lines and counted the occurrences of each one as shown in Figure 5.2, the results were compiled and the percentages calculated, this is shown in Figure 5.3.

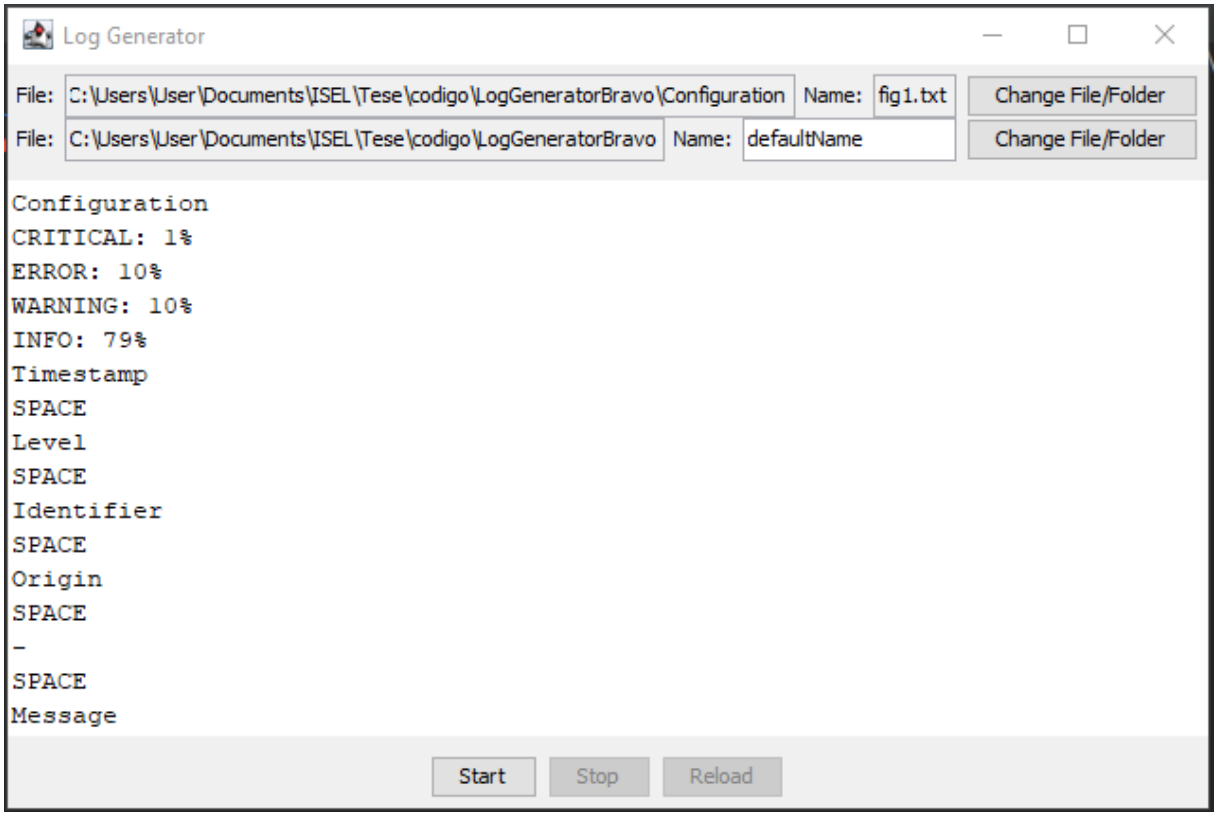


Figure 5.1: Log file generator, Configuration 1.

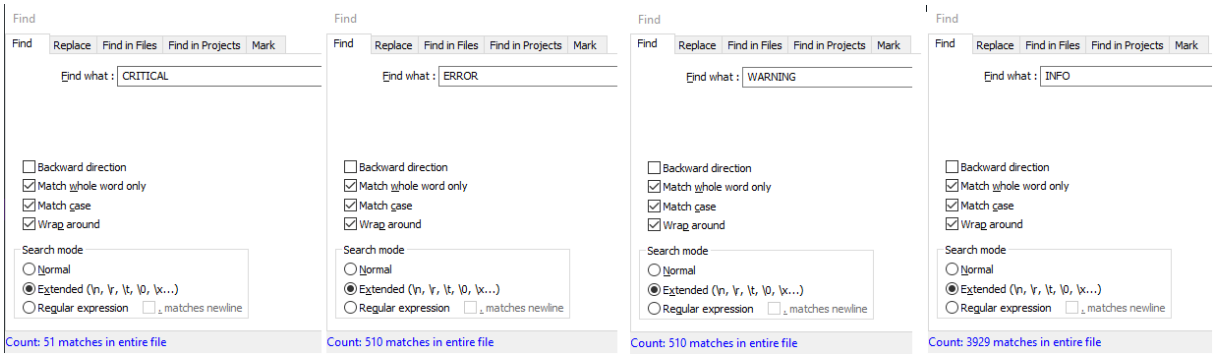


Figure 5.2: Individual levels count for Configuration 1.

	A	B	C
1	CRITICAL	51	0,0102
2	ERROR	510	0,102
3	WARNING	510	0,102
4	INFO	3929	0,7858
5	TOTAL	5000	1

Figure 5.3: Compiled results from level counts for Configuration 1.

The second configuration is shown in Figure 5.4. It has 7 levels. We generated a file with 4000 lines and counted the occurrences of each one as shown in Figure 5.5, the results were compiled and the percentages calculated, this is shown in Figure 5.6.

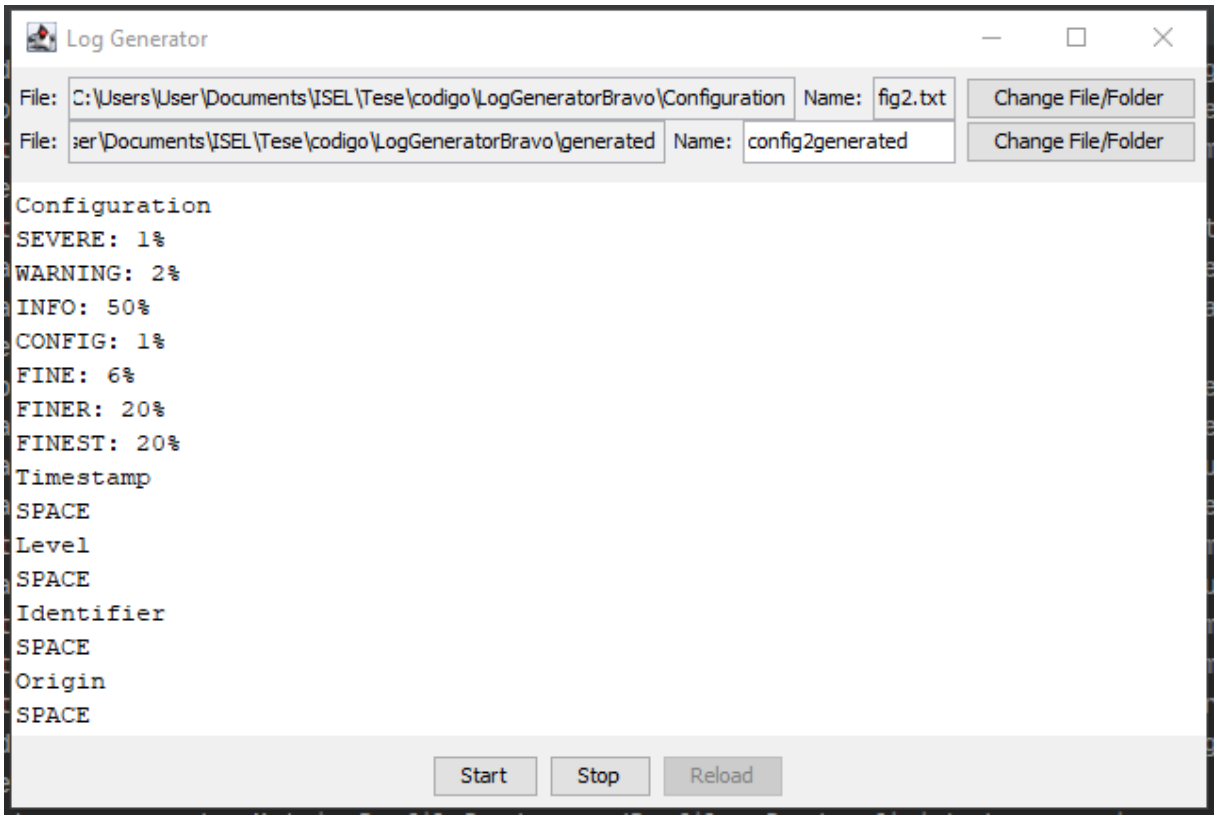


Figure 5.4: Log file generator, Configuration 2.

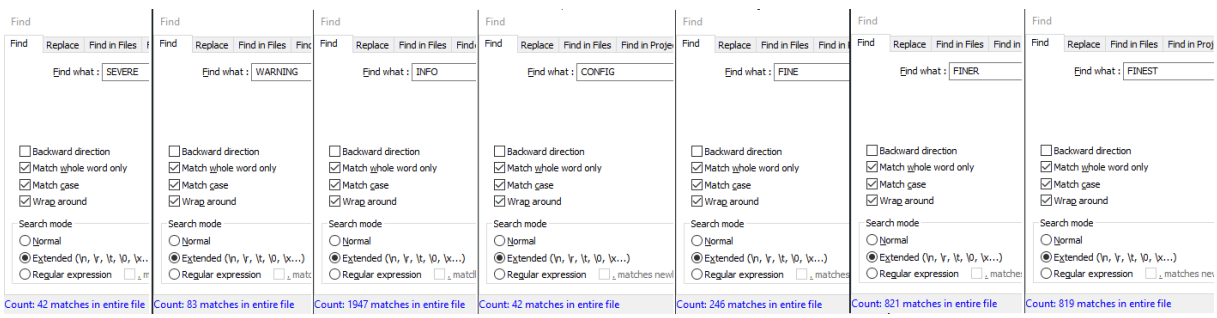


Figure 5.5: Individual levels count for Configuration 2.

	A	B	C
1	SEVERE	42	0,0105
2	WARNING	83	0,02075
3	INFO	1947	0,48675
4	CONFIG	42	0,0105
5	FINE	246	0,0615
6	FINER	821	0,20525
7	FINEST	819	0,20475
8	TOTAL	4000	1

Figure 5.6: Compiled results from level counts for Configuration 2.

As shown above, the log generation tool generates files according to the configuration in a reliable way. Figure 5.7 and Figure 5.8 show two examples of the tool we have developed displaying metrics on the files. In the figure, the section "Log level distribution" shows that the level distribution is the same as configured in the log generation tool.

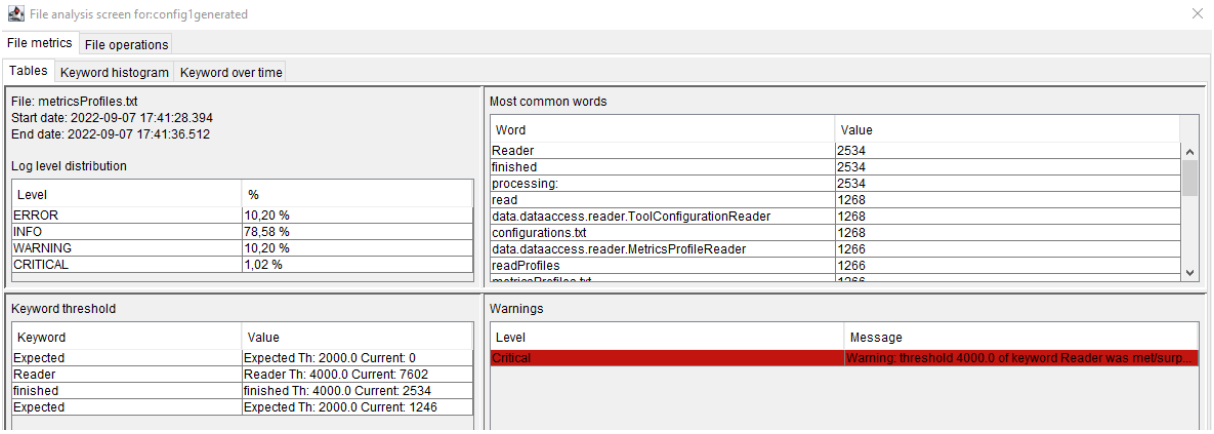


Figure 5.7: Log file analysis example for Configuration 1.

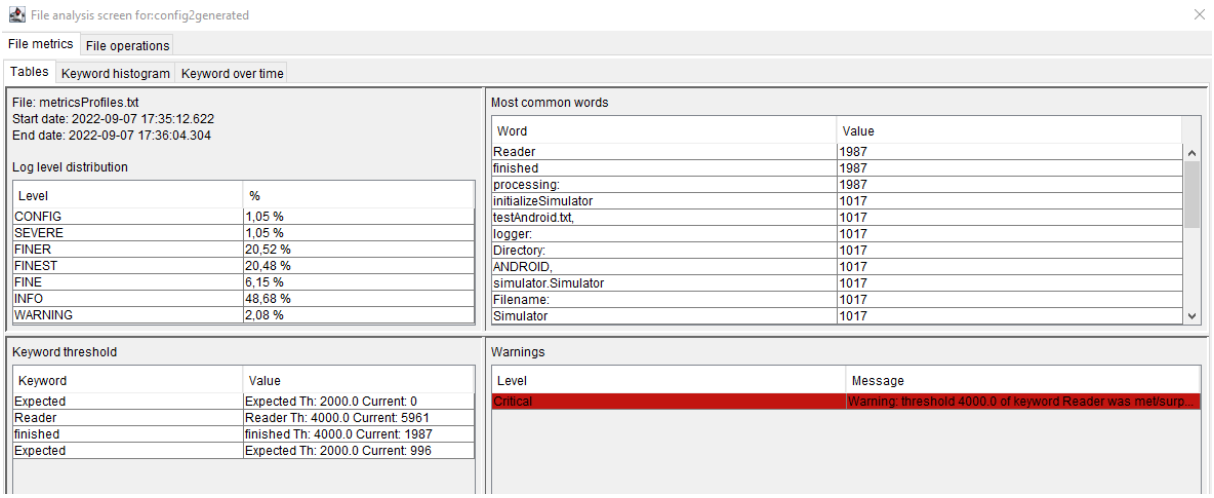


Figure 5.8: Log file analysis example for Configuration 2.

5.3 Log file searching

In this section, we will cover the processing of the log files with a focus on optimizing search times. Our tool processes the log files and transforms the data therein contained into a parsed format. When transforming data into other formats, there is a processing overhead and there may be a space overhead as well as in the final result. For example, applying a dictionary based compression to a 1kB file, the resulting file will be much larger than the original as the size of the dictionary has to be taken into account. With a dictionary with the size of 32kB then, the resulting file will likely be much larger than 1kB. A parallel may be drawn between a file compression algorithm and the parsing of the files in our tool. In order to have the data accessible, we load it into RAM. If we create an additional structure (the Suffix Tree, for example) we will need to occupy additional space. In terms of processing, reading and parsing the log line is a requirement, building the Suffix Tree takes additional processing and memory.

5.3.1 Effects of log file size - Suffix Trees

Based in the investigation done in Section 3.7 we concluded that applying a Suffix Tree would be the best approach for our scenario, in order to maximize search speed. However, in our initial tests with the application we faced an issue of lack of memory.

Table 5.1 presents the results of the time taken, comparing the processing of log files with and without the creation of the Suffix Tree. We created two files, one with 10 MB, and one with 100 MB of log lines.

Table 5.1: Metrics processing time with and without Suffix Trees

File size	Time in Milliseconds	Suffix Tree
10 MB	4607	No
10 MB	4104	No
10 MB	3887	No
10 MB	4039	No
10 MB	4260	No
10 MB	9780	Yes
10 MB	10266	Yes
10 MB	10119	Yes
10 MB	10206	Yes
10 MB	10456	Yes
100 MB	41249	No
100 MB	42694	No
100 MB	47243	No
100 MB	51144	No
100 MB	52758	No

Table 5.2: Processing time comparison for the algorithms with and without Suffix Trees

File size	Average in Milliseconds no ST	Average in Milliseconds with ST	Difference
10 MB	4179	10165	41.1%
20 MB	8555	Failure	-%
100 MB	47017	Failure	-%

From the results we obtained in Table 5.1 and Table 5.2, there is a substantial difference between creating the Suffix Trees or not. For the file with 100 MB the processing for the Suffix Trees caused an java.lang.OutOfMemoryError: Java heap space. As such we could not obtain processing times. We attempted as well to process a 20 MB file to see if with a smaller file the algorithm with Suffix Trees could work, but there was not enough memory.

What we conclude from this analysis is that even not having a limitation in terms of time, the memory required to create the Suffix Trees is prohibitive for large files. The gain one can have in searching is negated if we cannot build the actual trees. In the worst case scenario the space occupied by the Suffix Tree is $O(n^2)$.

5.3.2 Effects of log file size - Suffix Arrays

As using Suffix Trees was shown to be impossible, under our circumstances, we decided to use Suffix Arrays as their cost in terms of memory is much lower. For a string with m suffixes, we create an array for the indexes and the LCP array, occupying O(2m). This is much less memory intensive as compared to the Suffix Tree counterpart. The results of the processing are shown in Table 5.3

Table 5.3: Results processing time with and without Suffix Arrays

File size	Average in Milliseconds no SA	Average in Milliseconds with SA	Difference
10MB	2045	2377	16,24 %
20MB	3851	4831	25,44 %
50MB	9484	11629	22,61 %
100MB	18888	23146	22,54 %

5.3.3 Impact on search times

The research on Suffix Arrays and Suffix Trees was made in an attempt to reduce search times in the tool. We presented the consequences in terms of time and memory caused by the addition of support structures.

For our tests we used a file with a large string in the message part of the log line (47166 characters), and a file with an average expected message size of 100 characters with 100MB and over 750k lines. This is meant to reduce the risk of one algorithm performing much better than another based only on the size of the strings.

We tested three search algorithms, one without a support structure, which reads the string using indexOf() counting the located indexes, and two algorithms based in Suffix Arrays. For the Suffix Array algorithms we used an iterative and a recursive implementation of the binary search. We ran the same test twenty times for each algorithm and for each file. The results are reported in Table 5.4 and Table 5.5.

The results showed that the search made without a search structure is consistently quicker than using a support structure like a Suffix Array, and that the iterative implementation is faster than the recursive one. This can be explained by the overhead caused by the repeated function calls [21]. Besides possible mistakes in our implementation of the algorithm, what we conclude is that already existing search algorithms in Java are optimized the best they can be.

Table 5.4: Search time results with a large message file

Without Support Structure	Suffix Array Iterative Search	Suffix Array Recursive Search
Time in milliseconds	Time in milliseconds	Time in milliseconds
1,0883	1,4523	1,4195
0,2859	0,4724	0,4733
0,2716	0,4628	0,4616
0,2919	0,467	0,3333
0,1527	0,4141	0,4222
0,2052	0,4128	0,4099
0,2289	0,4102	0,4249
0,1529	0,3103	0,3579
0,2155	0,3459	0,4106
0,1819	0,4155	0,4175
0,2158	0,446	0,2811
0,1349	0,415	0,4249
0,2211	0,4241	0,4362
0,1342	0,4261	0,4152
0,2184	0,4237	0,4179
0,1692	0,4059	0,4183
0,1372	0,409	0,433
0,2207	0,4176	0,4175
0,2228	0,4136	0,3693
0,1332	0,4194	0,3661
Average time	Average time	Average time
0,244115	0,468185	0,45551

Table 5.5: Average search time results for 100 MB file size with average message size. The complete table can be found in Appendix B.

Without Support Structure	Suffix Array Iterative Search	Suffix Array Recursive Search
Average time in milliseconds	Average time in milliseconds	Average time in milliseconds
79,943375	233,77235	260,56162

Figure 5.9 displays two plotted charts with the times reported in Table 5.4 and Table 5.5. In these we can visually see the reduction in time from the first processing in

comparison to the following searches. Additionally, there seems to be a pattern in the processing of the files, even using different implementations of the algorithms the shape the graphs exhibit are similar.

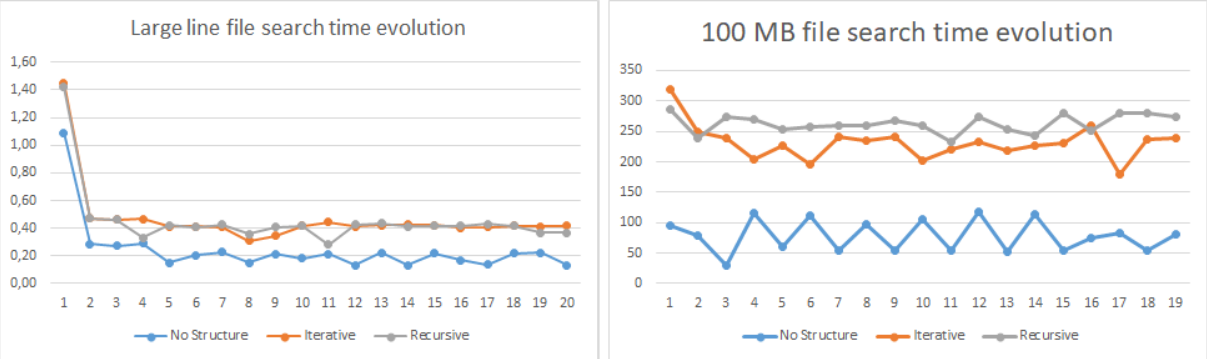


Figure 5.9: Evolution of searching performance in both files.

5.3.4 Search time and memory with support structures

The premise of adding support structures like Suffix Arrays to our search algorithm was to reduce search times. From the results we obtained, the structure which can be used for this purpose is a Suffix Array, as Suffix Trees consume too much memory. The creation of Suffix Arrays consumes approximately 22% more time than not using any support structure. Additionally, in spite of having this structure, the actual search time is worse than without it.

In the test made with a 100MB file, we found the expected 1517080 matches of the word in question, in approximately 80 milliseconds without using a support structure. We took advantage of parallel processing, which was done for all algorithms, processing each log line in parallel. We conclude that there is no need for complex structures since reading a file with over 750k lines is done in a timely manner.

5.4 Log file analysis

In this section, we will present results demonstrating the accuracy of the metrics employed. For a baseline comparison, we will use Notepad++ [36] open source text editor which is widely used.

For this analysis using the log generation tool we generated a file with the configurations shown in Figure 5.10. This log file will have the log level distribution shown in the configuration. The messages are one of four possibilities, chosen at random, as

the contents of the messages themselves matter very little, since we are not analysing patterns.

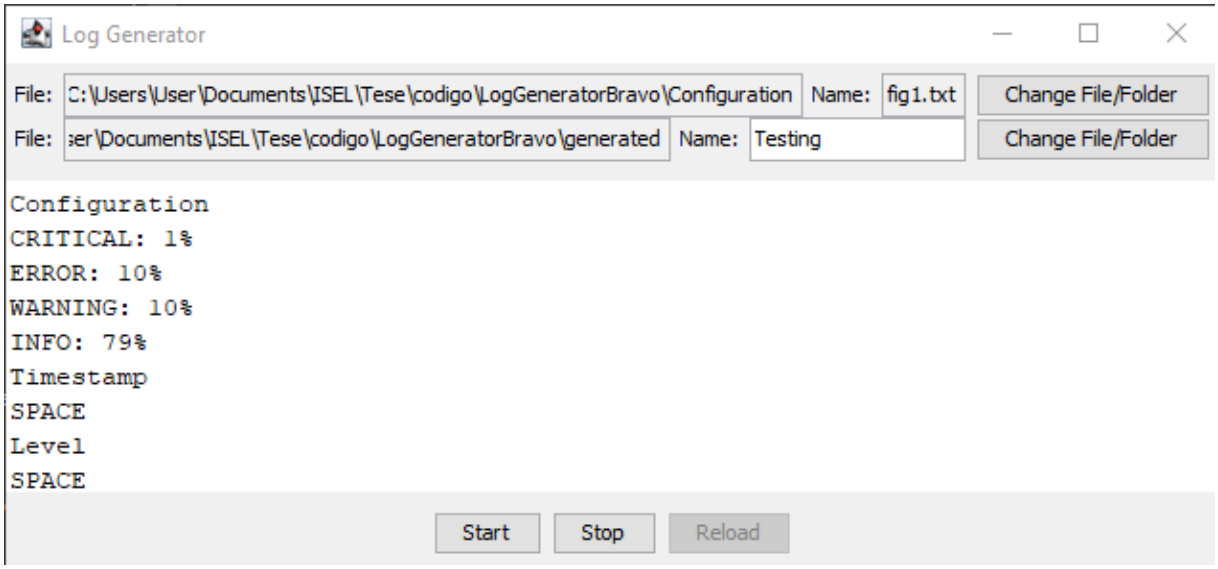


Figure 5.10: Log file generator configuration for test.

The distribution of log levels in the generated log file was as specified. In Figure 5.11 we show the table calculated in a spreadsheet, and on the right the actual result of the application.

Level	Occurrences	Percentage	Percentage
WARNING	999	0,1011	10%
ERROR	999	0,1011	10%
INFO	7787	0,7878	79%
CRITICAL	100	0,0101	1%
	9885		

File: metricsProfiles.txt
Start date: 2022-07-15 22:05:02.005
End date: 2022-07-15 22:05:02.986

Log level distribution

Level	%
ERROR	10,11 %
INFO	78,78 %
WARNING	10,11 %
CRITICAL	1,01 %

Figure 5.11: Distribution of the log levels.

In Figure 5.12, we present the results of the most common words analysis. On the top we have the table in the tool, and below it, four examples taken using Notepad++. We restricted the search in Notepad++ to matching the whole word (so if the word exists within another it is not counted as a single word) and as case sensitive.

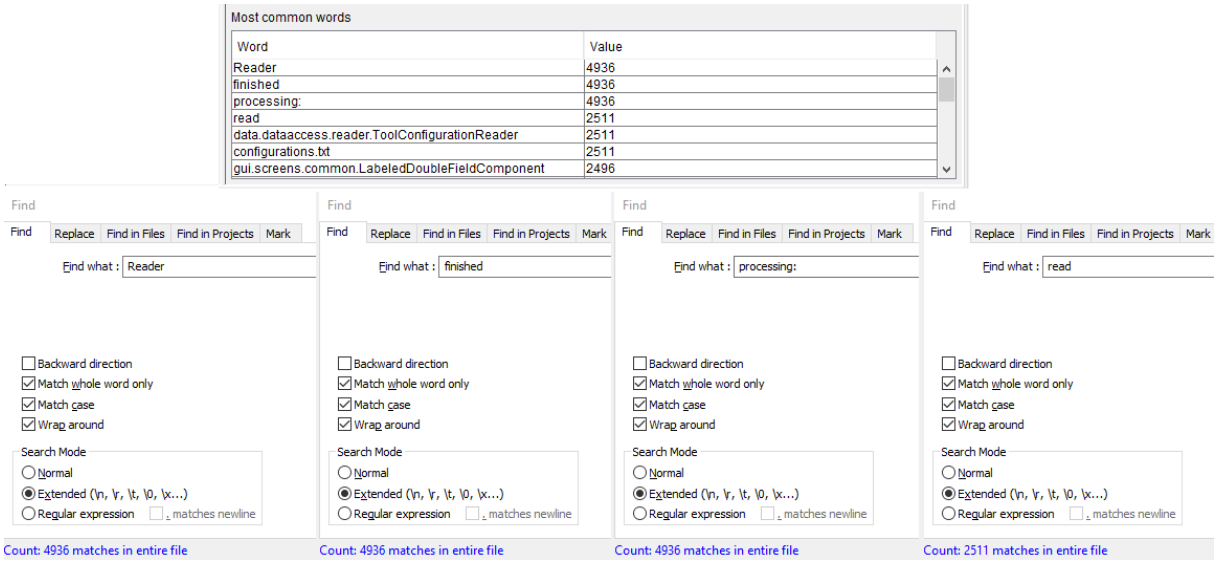


Figure 5.12: Most common words in the tool and some examples extracted from Notepad++.

For testing the thresholds, we defined the metrics profile present in Figure 5.13. There we defined four thresholds, "Reader" is not case sensitive and will trigger at 4000 occurrences or higher. The string "finished" is case sensitive and will trigger at 2000 occurrences or higher, "Expected" will trigger at 2000 occurrences or higher, for this one we created a case sensitive and a non case sensitive keyword configuration.

In Figure 5.14 we present the results. The results extracted from Notepad++ follow the same configuration defined in the threshold. As such, we can see that the "Expected" in case sensitive has no matches and the one in non case sensitive has more than 2000.

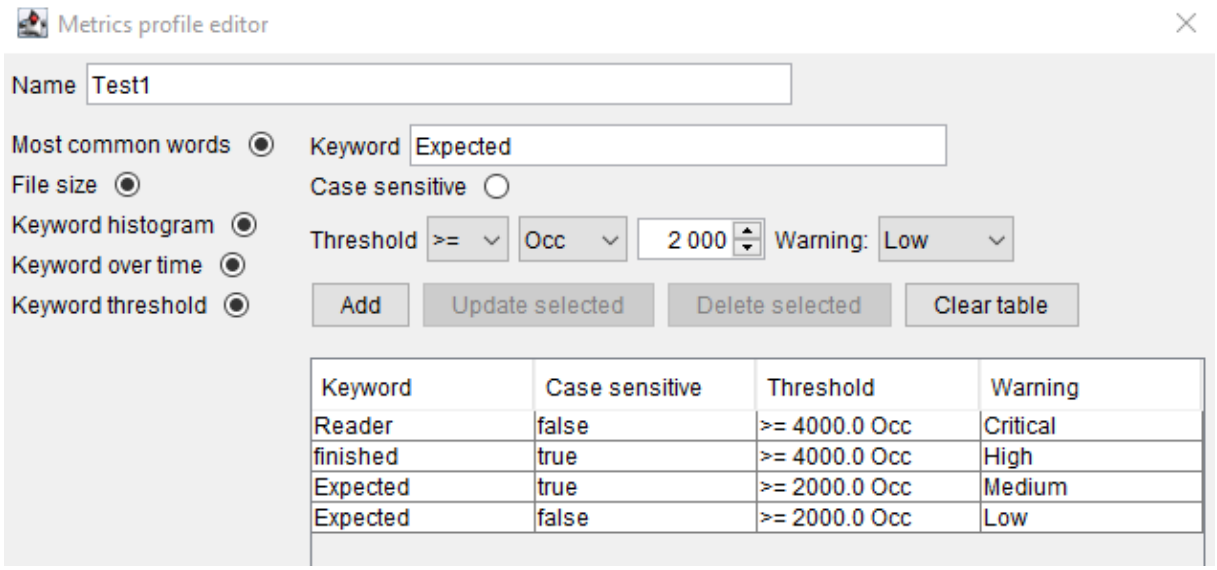


Figure 5.13: Metrics profile used for keyword thresholds.

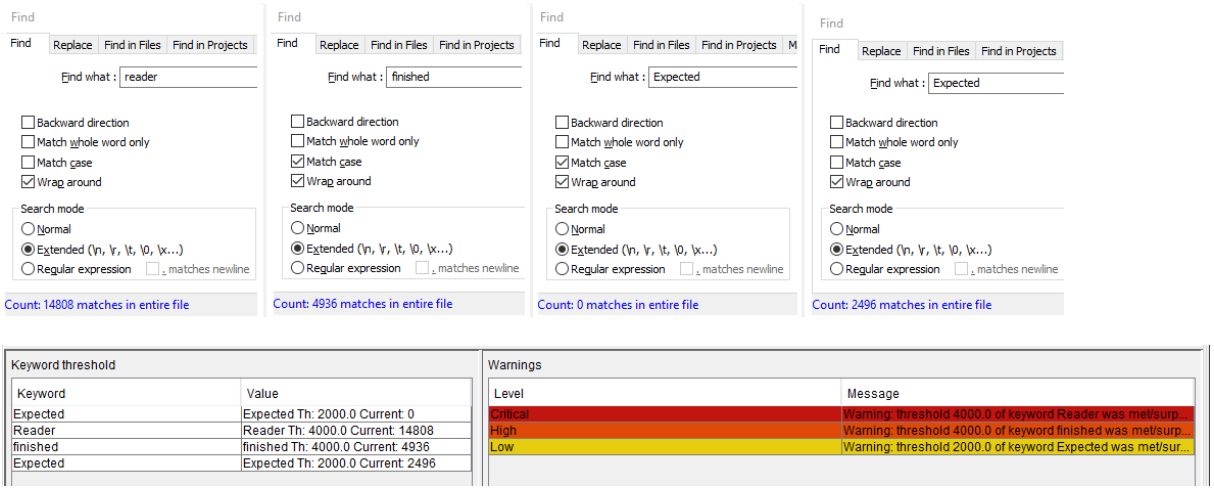


Figure 5.14: Occurrences counted in Notepad++, keyword thresholds, and generated warnings.

We present the histogram composed by the keywords in Figure 5.15. We can see that one of the keywords is much more predominant than the other two and the histogram counts are in accordance with Figure 5.14. In Figure 5.16 we show the evolution of one of the keywords over time, in this case the evolution was linear.

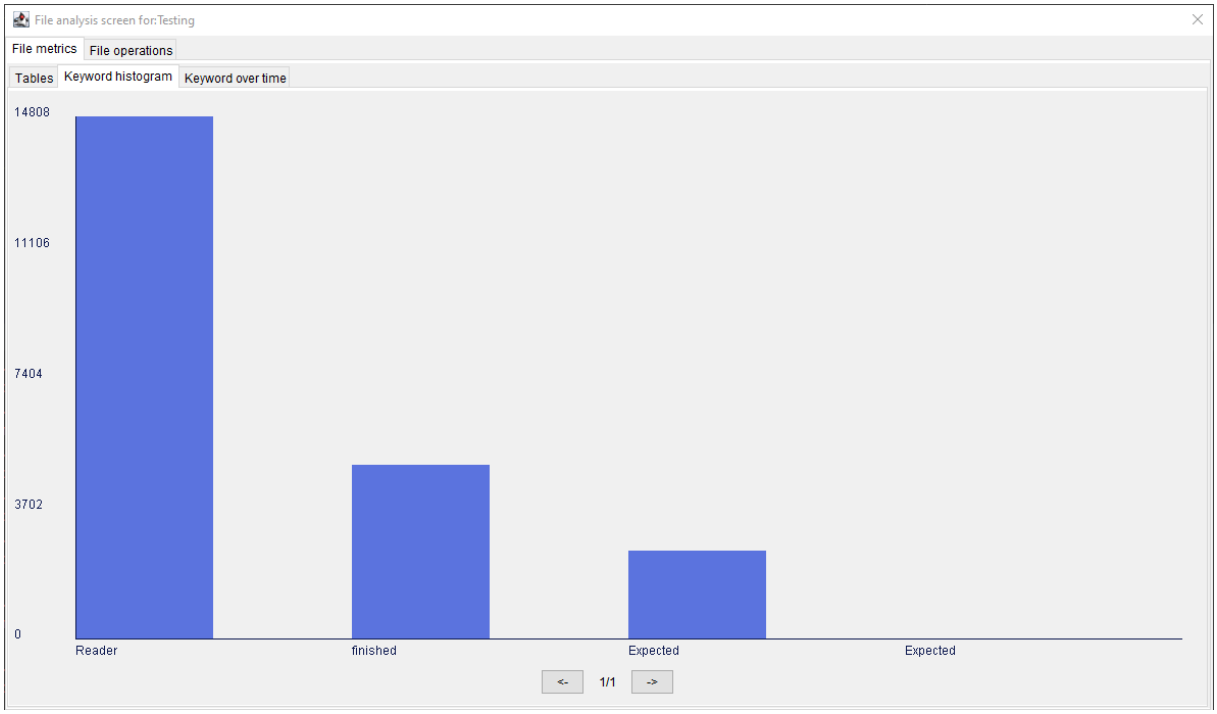


Figure 5.15: Keyword histogram example.

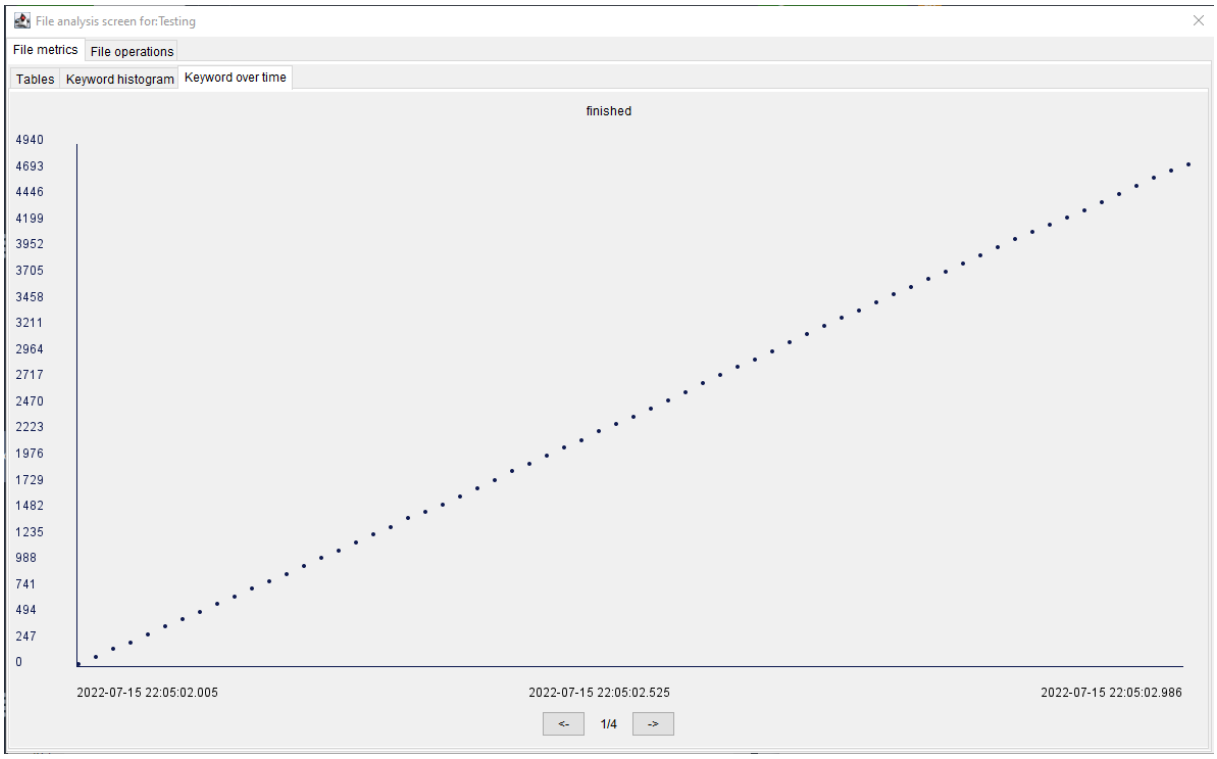


Figure 5.16: Keyword over time example.

5.5 Log file monitoring

For this test we used the log generation tool as well, we defined a metrics profile with the thresholds displayed in Figure 5.17. We defined four thresholds, "Reader" is not case sensitive and will trigger at 40 occurrences or higher. "finished" is case sensitive and will trigger at 20 occurrences or higher, "Expected" will trigger at 20 occurrences or higher, for this one we created a case sensitive and a non case sensitive keyword configuration.

In Figure 5.18 we present the results. In this figure we present several snapshots of the monitoring results, and the evolution of the generated warnings as the thresholds were surpassed or met. At the bottom of the image we present the most common words, which help illustrate the words we were looking for in the thresholds.

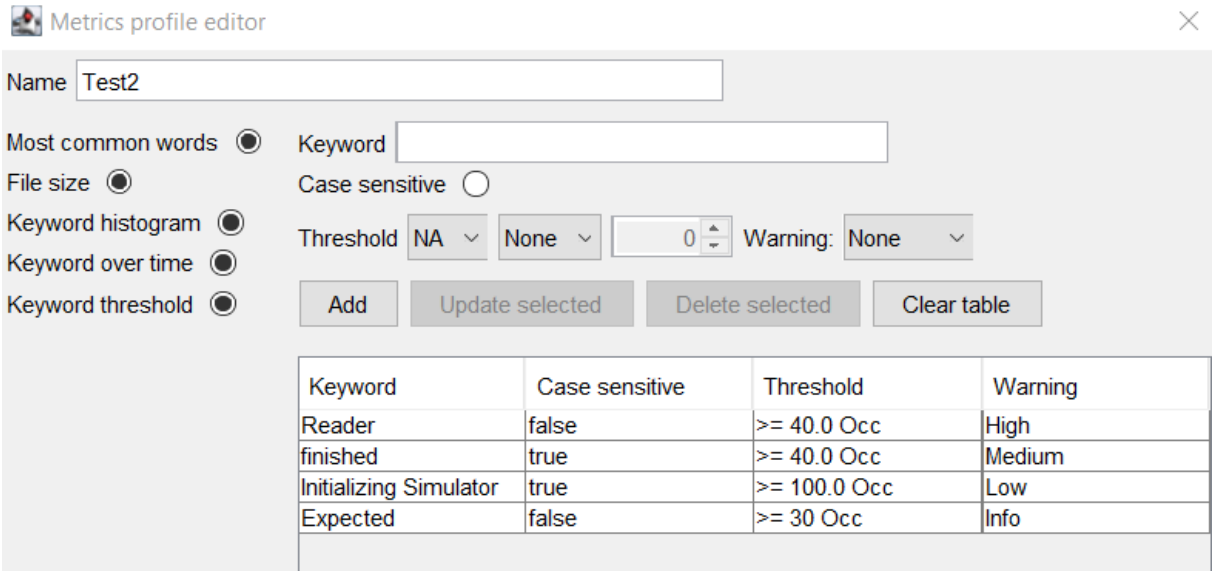


Figure 5.17: Metrics profile used in monitoring demonstration.

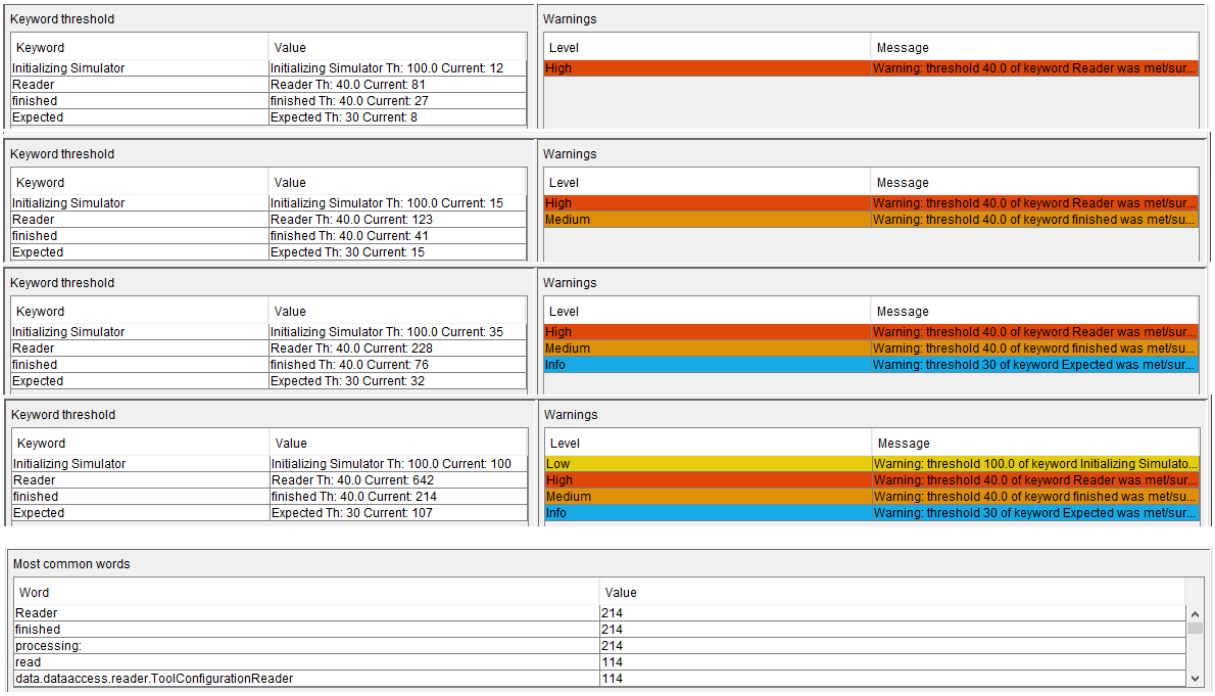


Figure 5.18: Keyword warnings example.

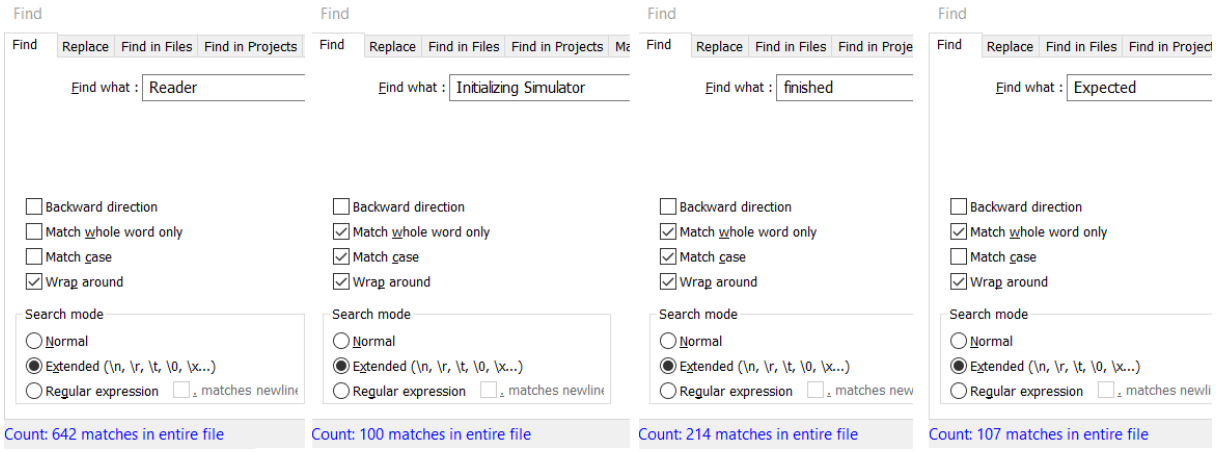


Figure 5.19: Occurrences counted in Notepad++.

In Figure 5.20 we present the resulting histogram for the keywords in the metrics profile. Figure 5.21 shows the evolution of one of the keywords over time, in this case the evolution was linear.

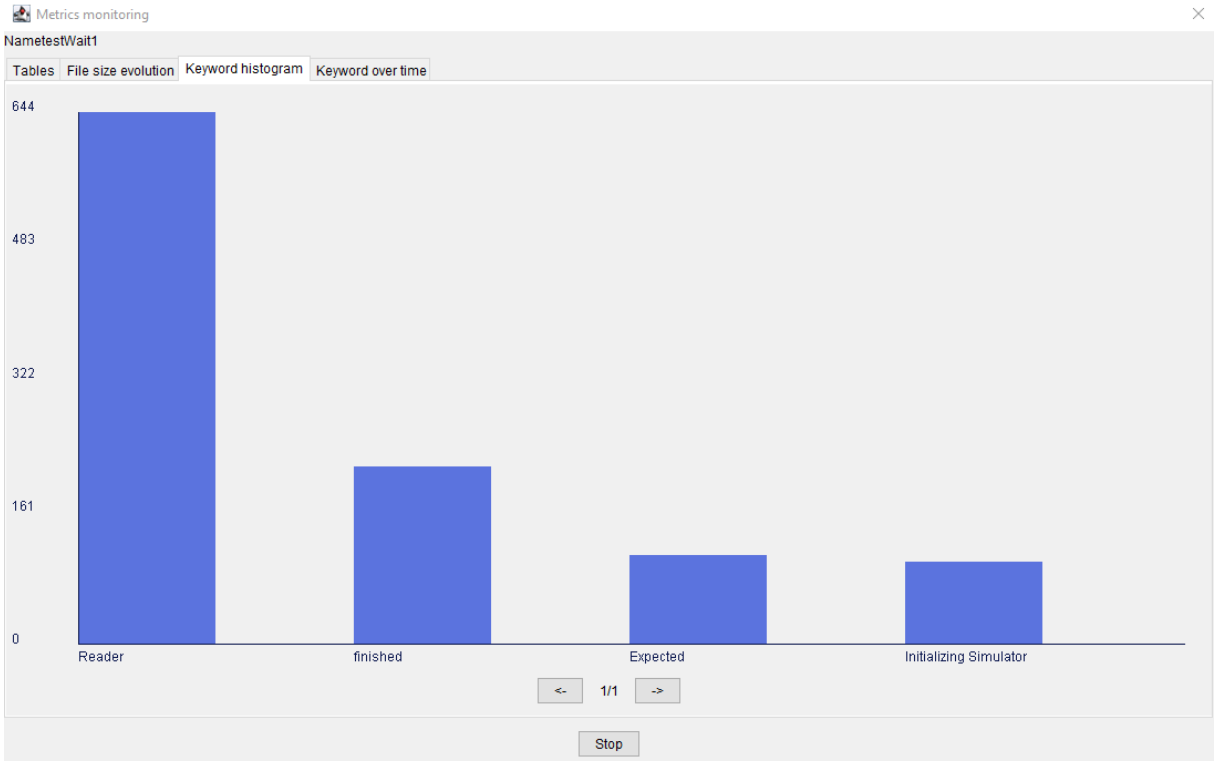


Figure 5.20: Keyword histogram example.

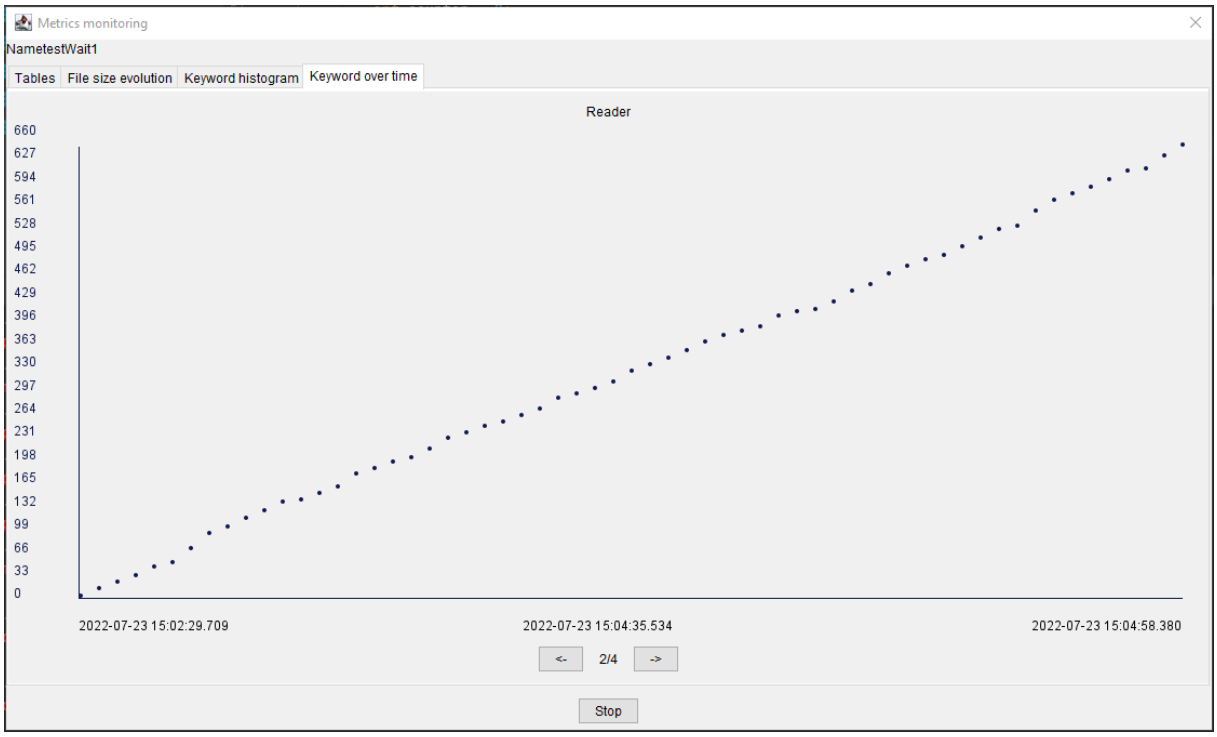


Figure 5.21: Keyword over time example.

In Figure 5.22 we present the comparison between the monitoring of the file size and the actual file size, extracting the results from Notepad++.

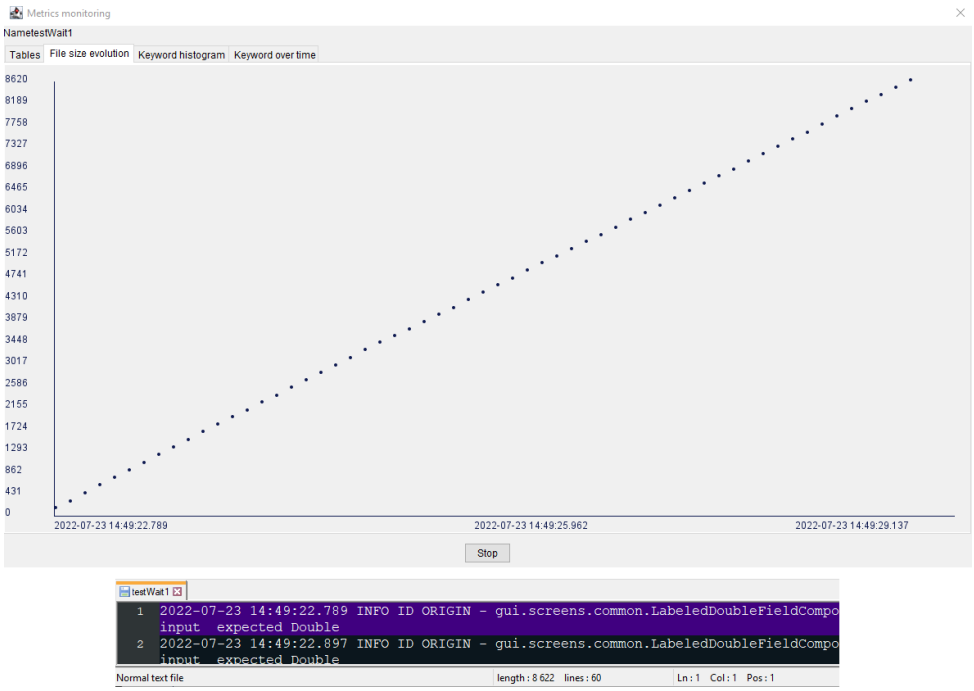


Figure 5.22: File size evolution example.

5.6 File operations

There are three main features in the file operations. They are Searching, Filtering, and Exporting. We show an example for each of them, the file was generated with the log generation tool.

Figure 5.23 demonstrates a search operation done for the string "Reader". The search is case sensitive, as such, the "reader" in lower-case is not highlighted. The current position of the search is highlighted in a darker color. Pressing "Next" and "Previous" will move the highlighted area to the corresponding position. If you reach the end of the file, pressing "Next" will move the highlighted area to the to the first occurrence.

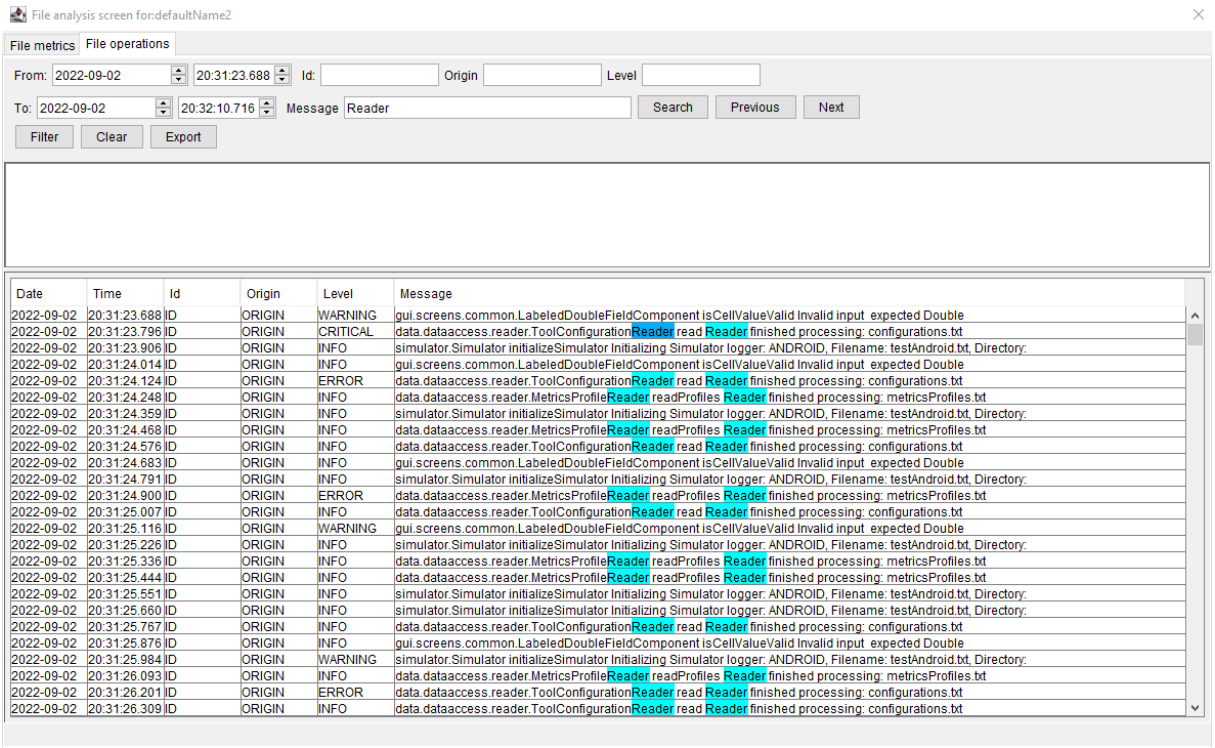


Figure 5.23: Search example.

In Figure 5.24, we display the results of filtering. We used the same search keyword as in the previous example. Additionally, we filtered by the log level "ERROR". Comparing this figure with the previous one, we can see the differences in the output between searching and filtering.

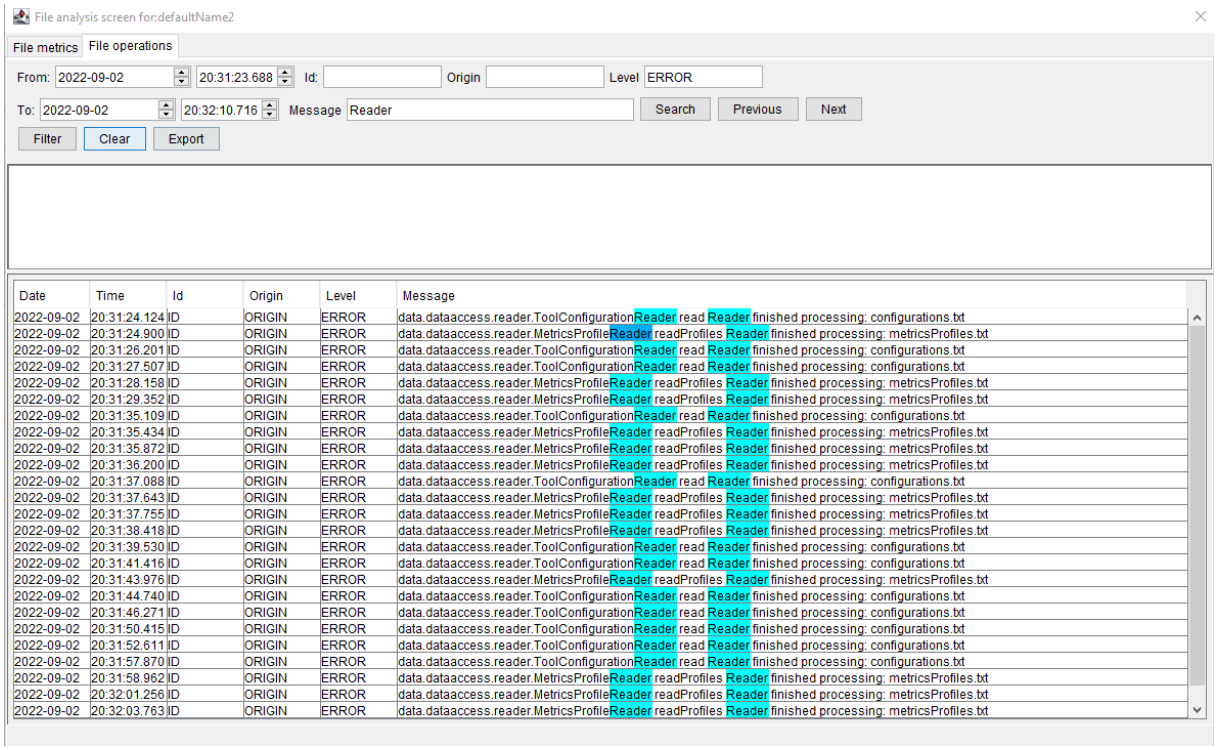


Figure 5.24: Filtering example.

The export functionality is displayed in Figure 5.25. This figure displays the exported file and the filtered data, as shown, the file contains the same amount of lines and the same contents. The highlighted text in both applications is different, as one is case sensitive and the other is not and is only highlighting the isolated word.

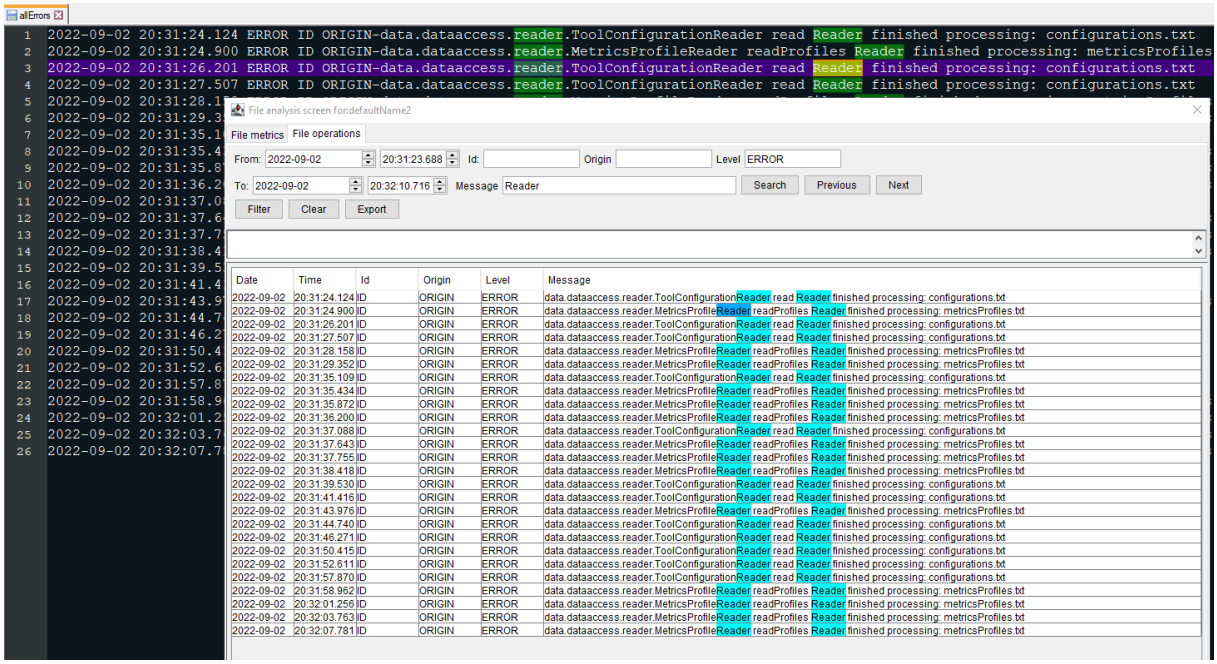


Figure 5.25: Export example.

5.7 Organization

For the organization feature we created a simple metrics profile shown in Figure 5.26. There are two thresholds, one for "Reader" as case sensitive in which it will trigger after 5 occurrences. The second threshold is for the keyword "test" as non case sensitive, this threshold will trigger if the keyword is more or equal to 10% of all words in the messages. In order to test these two thresholds, we created four files, shown in Figure 5.27 and Figure 5.28. For the test we selected all four files, but only two are meant to reach the thresholds.

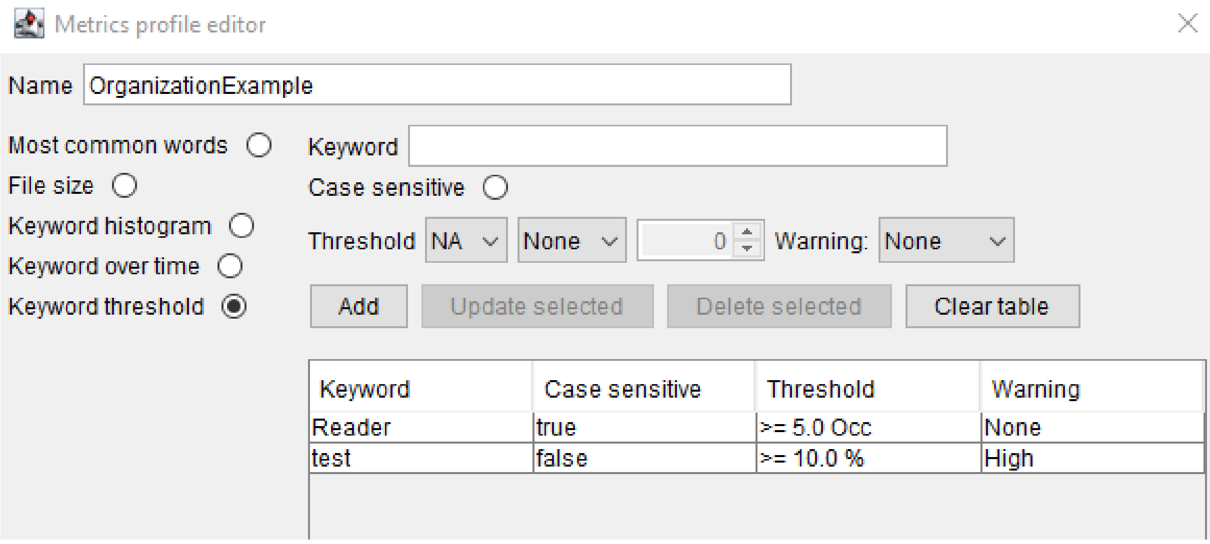
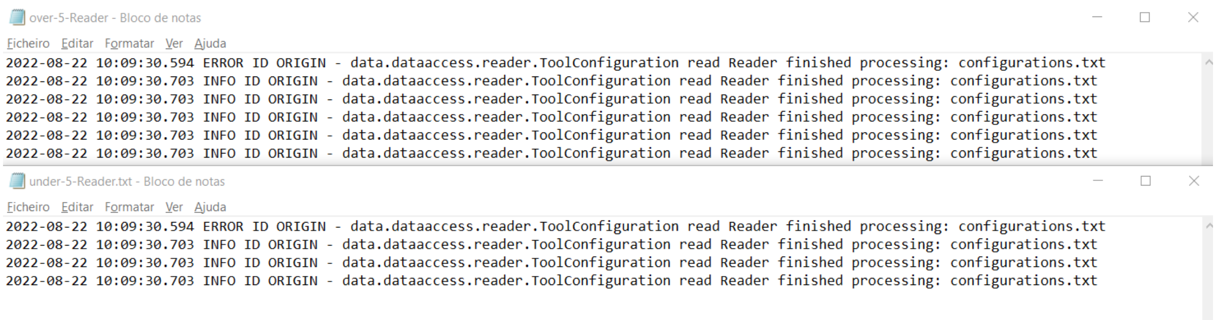


Figure 5.26: Metrics profiles created for the organization example.




```
2022-08-22 10:09:30.594 ERROR ID ORIGIN - data.dataaccess.reader.ToolConfiguration read Reader finished processing: configurations.txt
2022-08-22 10:09:30.703 INFO ID ORIGIN - data.dataaccess.reader.ToolConfiguration read Reader finished processing: configurations.txt
2022-08-22 10:09:30.703 INFO ID ORIGIN - data.dataaccess.reader.ToolConfiguration read Reader finished processing: configurations.txt
2022-08-22 10:09:30.703 INFO ID ORIGIN - data.dataaccess.reader.ToolConfiguration read Reader finished processing: configurations.txt
2022-08-22 10:09:30.703 INFO ID ORIGIN - data.dataaccess.reader.ToolConfiguration read Reader finished processing: configurations.txt
2022-08-22 10:09:30.703 INFO ID ORIGIN - data.dataaccess.reader.ToolConfiguration read Reader finished processing: configurations.txt
```



```
2022-08-22 10:09:30.594 ERROR ID ORIGIN - data.dataaccess.reader.ToolConfiguration read Reader finished processing: configurations.txt
2022-08-22 10:09:30.703 INFO ID ORIGIN - data.dataaccess.reader.ToolConfiguration read Reader finished processing: configurations.txt
2022-08-22 10:09:30.703 INFO ID ORIGIN - data.dataaccess.reader.ToolConfiguration read Reader finished processing: configurations.txt
2022-08-22 10:09:30.703 INFO ID ORIGIN - data.dataaccess.reader.ToolConfiguration read Reader finished processing: configurations.txt
```

Figure 5.27: Files used for the occurrences threshold.

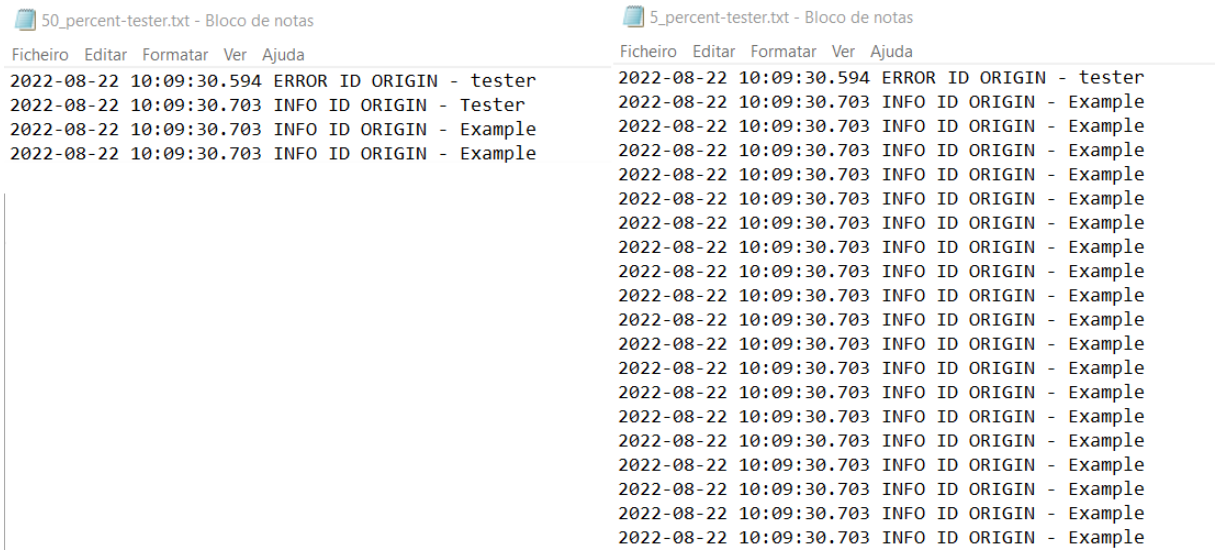


Figure 5.28: Files used for the percentage threshold.

Figure 5.29 displays the setup for the demonstration. The four files were selected, the appropriate parsing profile and metrics profile were selected.

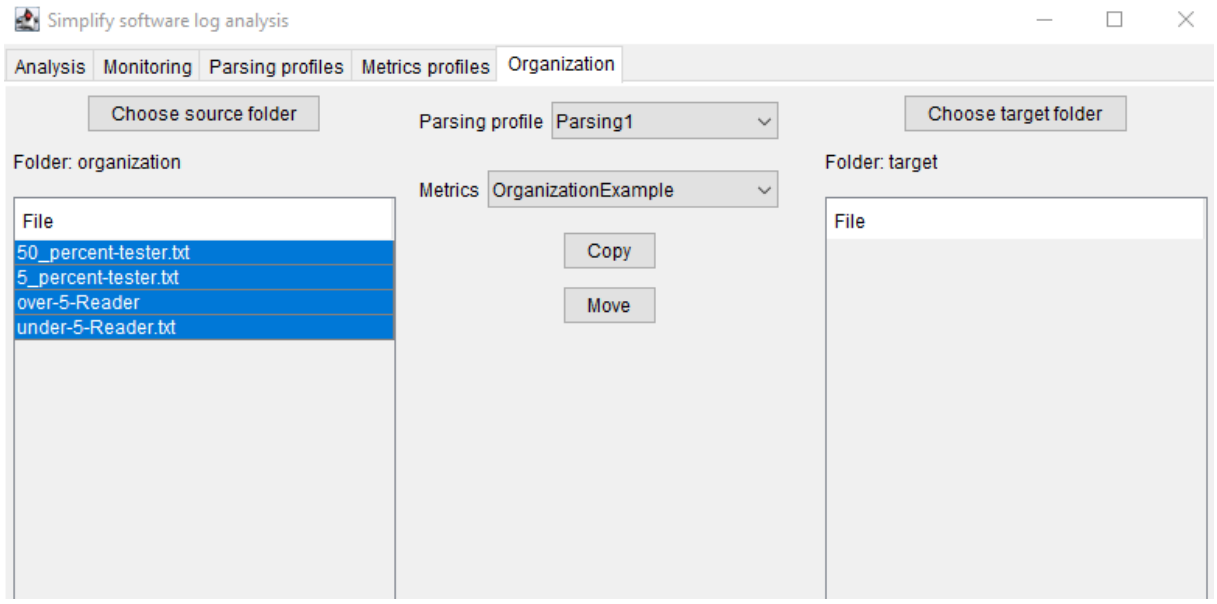


Figure 5.29: Organization screen configuration.

As we can see from the results, only the file with 50% of the keywords being "test" and the file in which there are over 5 instances of "Reader" were copied to the target folder.

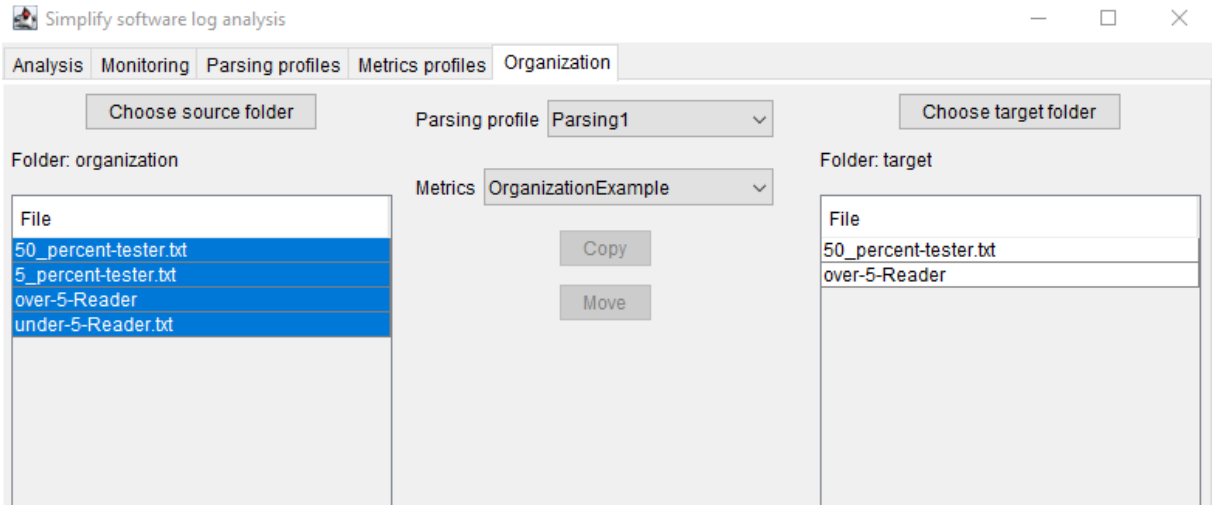


Figure 5.30: Organization operation results.

5.8 Results

We demonstrated that the log generation tool is working, which enables us to have more versatile tests. In order to better understand the consequences of file size in the time needed to process it, as well as the consequences of using a support structure or not several tests were made, in which we concluded that using Suffix Trees was not viable due to the excessive memory use. Following that find, we tested the same effects but using Suffix Arrays, to which we concluded that in terms of memory, the Suffix Arrays were viable. However, we then used the Suffix Arrays to search the files and compared the times to using a simple search algorithm with no support structure, the search times without the use of Suffix Arrays were better. We concluded that, for our purposes, the usage of support structures is detrimental.

The remainder of the chapter is a demonstration of the features, we present all features for log file analysis, log file monitoring with proof based on the usage of Notepad++ [36]. We demonstrated the file operations (search, filtering and export) as well as the file organization. Covering this way all of the features we had defined in Section 1.2.

6

Conclusions

Logging is an integral part of digital systems. Their development, maintenance and support are built upon the information and context they provide. Log files can be quite extensive and analysing them is a time consuming, albeit necessary task.

The field of study on logging is vast, covering ways to improve the quality of logs, how to extract information from them, ways of dynamically adjusting logging verbosity among other topics. For our needs we focused on the parsing aspect. There are several tools in the field which provide support for users to analyse and monitor log files and extract information from them. In this work, we aimed to provide a less complex alternative for the common user.

In this work, we have developed a tool which, through user provided configuration, parses the log files. The information obtained is then processed and displayed providing a summary on the file, some relevant statistics as well as providing the user with search, filtering, and exporting capabilities. Another application of the parsed information is real time monitoring of the file's contents with a configurable dashboard.

The tool was tested, both in its monitoring and analysis capacities and the file search algorithms were tested as well. We concluded that all the features were created as proposed, and that the simplest search algorithm was the one with the best results for our needs.

The features were delivered as indicated, the usefulness of said features and improvements on their functioning would have to be tested with real exposure to a wide range of users. From our experience, its the users that decide the quality of the tool.

As we decided to create a tool to be available as open source, we faced the issue of not knowing the limitations, both legal and practical of using other open source code. Additionally, we decided to take full control of the code that is used in the tool, in order to limit exposure to third party dependencies, bugs, and vulnerabilities. We stand by that decision, however, it has limited our capability of creating more functional and more aesthetically pleasing charts and graphs. As another valid approach, we could have opted to use other open source libraries, such as JFreeChart [25].

6.1 Future work

As part of future work, the tool needs an improvement in the charts regarding the visual quality of the information displayed.

We had also set out with the idea of potentially adding machine learning to the tool, taking advantage of the parsed data. With machine learning and training classifiers we could, for example, provide a richer event recognition system for the monitoring features.

References

[1] Shilin He, Pinjia He, Zhuangbin Chen, Tianyi Yang, Yuxin Su, and Michael R. Lyu, *A Survey on Automated Log Analysis for Reliability Engineering*. 2021. arXiv: 2009.07237 [cs.SE].

[2] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan, *Detecting large-scale system problems by mining console logs*. 2009, pages 117–132.

[3] Andreas Hotho, Andreas Nürnberger, and Gerhard Paaß, "A brief survey of text mining.", in *Ldv Forum*, vol. 20, 2005, pages 19–62.

[4] Sina Gholamian and Paul AS Ward, "A comprehensive survey of logging in software: From logging statements automation to log mining and analysis", *arXiv preprint arXiv:2110.12489*, 2021.

[5] Ali Intezari, David J Pauleen, and Nazim Taskin, "The DIKW hierarchy and management decision-making", in *49th Hawaii International Conference on System Sciences (HICSS)*, IEEE, 2016, pages 4193–4201.

[6] Wikipedia. "Aho–corasick algorithm". (Sep. 2022), [Online]. Available: https://en.wikipedia.org/wiki/Aho-Corasick_algorithm.

[7] Alfred V Aho and Margaret J Corasick, "Efficient string matching: An aid to bibliographic search", *Communications of the ACM*, vol. 18, no. 6, pages 333–340, 1975.

[8] E. Alpaydin, *Introduction to machine learning*, 2nd. The MIT Press, 2010.

[9] AWS. "What is amazon corretto 11?" (Sep. 2022), [Online]. Available: https://docs.aws.amazon.com/corretto/latest/corretto-11-ug/what-is-corretto-11.html.

[10] Robert S Boyer and J Strother Moore, "A fast string searching algorithm", *Communications of the ACM*, vol. 20, no. 10, pages 762–772, 1977.

[11] IETF. "Rfc4180". (Sep. 2022), [Online]. Available: https://datatracker.ietf.org/doc/html/rfc4180.

[12] Data Dog. "Data dog product". (Sep. 2022), [Online]. Available: https://www.datadoghq.com/product/.

[13] John Demian. "Why log management is so important". (Sep. 2022), [Online]. Available: https://dzone.com/articles/why-is-log-management-so-important-and-how-can-it.

[14] Artur Ferreira, Arlindo Oliveira, and Mário Figueiredo, "On the use of suffix arrays for memory-efficient Lempel-Ziv data compression", in *2009 Data Compression Conference*, IEEE, 2009, pages 444–444.

[15] Fluentd. "What is fluentd?" (Sep. 2022), [Online]. Available: https://www.fluentd.org/architecture.

[16] T. Hastie, R. Tibshirani, and J. Friedman, *The elements of statistical learning*, 2nd. Springer, 2009.

[17] Google. "Glog repository". (Sep. 2022), [Online]. Available: https://github.com/google/glog.

[18] ——, "Glog repository logging.h". (Sep. 2022), [Online]. Available: https://github.com/google/glog/blob/master/src/glog/logging.h.in.

[19] ——, "User guide". (Sep. 2022), [Online]. Available: https://github.com/google/glog#user-guide.

[20] IBM. "What is three-tier architecture?" (Sep. 2022), [Online]. Available: https://www.ibm.com/cloud/learn/three-tier-architecture#toc-what-is-th-tyaDx4QJ.

[21] Techdifferences. "Difference between recursion and iteration". (Sep. 2022), [Online]. Available: https://techdifferences.com/difference-between-recursion-and-iteration-2.html.

[22] Oracle. "How to set the look and feel". (Sep. 2022), [Online]. Available: https://docs.oracle.com/javase/tutorial/uiswing/lookandfeel/plaf.html.

[23] ——, "Class pattern". (Sep. 2022), [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html.

[24] ——, "Interface serializable". (Sep. 2022), [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html.

[25] JFreeChart. "Welcome to jfreechart!" (Sep. 2022), [Online]. Available: https://www.jfree.org/jfreechart/.

[26] Json. "Introducing json". (Sep. 2022), [Online]. Available: https://www.json.org/json-en.html.

[27] Apache Software Foundation. "Configuration". (Sep. 2022), [Online]. Available: https://logging.apache.org/log4j/2.x/manual/configuration.html.

[28] Log4Net. "What is apache log4net™". (Sep. 2022), [Online]. Available: http://logging.apache.org/log4net/index.html.

[29] ——, "Apache log4net™ frequently asked questions". (Sep. 2022), [Online]. Available: http://logging.apache.org/log4net/release/faq.html.

[30] Loggly. "Visualize, analyze, inspect, and solve." (Sep. 2022), [Online]. Available: https://www.loggly.com/product/.

[31] Microsoft. "The transaction log (SQL server)". (Sep. 2022), [Online]. Available: https://docs.microsoft.com/en-us/sql/relational-databases/logs/the-transaction-log-sql-server?view=sql-server-ver15.

[32] Oracle. "Transaction log settings". (Sep. 2022), [Online]. Available: https://docs.oracle.com/cd/E50612_01/doc.11122/user_guide/content/log_global_settings.html.

[33] PostgreSQL Global Development Group. "Write-ahead logging (WAL)". (Sep. 2022), [Online]. Available: https://www.postgresql.org/docs/9.1/wal-intro.html.

[34] Logstash. "Centralize, transform & stash your data". (Sep. 2022), [Online]. Available: https://www.elastic.co/logstash/.

[35] C. Manning, P. Raghavan, and H. Schütze, *Introduction to information retrieval*. Cambridge University Press, 2008.

[36] Don Ho. "Notepad++ homepage". (Sep. 2022), [Online]. Available: https://notepad-plus-plus.org/.

[37] Klaus K Obermeier, "Grok—a knowledge-based text processing system", in *Proceedings of the ACM fourteenth annual conference on Computer science*, 1986, pages 331–339.

[38] Adam Oliner, Archana Ganapathi, and Wei Xu, "Advances and challenges in log analysis: Logs contain a wealth of information for help in managing systems.", *Queue*, vol. 9, no. 12, pages 30–40, 2011.

[39] Python Software Foundation. "Logging — logging facility for python". (Sep. 2022), [Online]. Available: https://docs.python.org/3/library/logging.html.

[40] Vinay Sajip. "Logging howto". (Sep. 2022), [Online]. Available: https://docs.python.org/3/howto/logging.html#logging-basic-tutorial.

[41] Stuart Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach*, 3rd. Prentice Hall, 2010.

[42] Gary Glover. "A watchtower is pointless if there's no watchman inside". (Sep. 2022), [Online]. Available: https://www.securitymetrics.com/blog/importance-log-management.

[43] Serilog. "Flexible, structured events — log file convenience." (Sep. 2022), [Online]. Available: https://serilog.net/.

[44] SQLite. "What is sqlite?" (Sep. 2022), [Online]. Available: https://www.sqlite.org/index.html.

[45] Pang Ko and Srinivas Aluru, "Suffix tree applications in computational biology", *Handbook of Computational Molecular Biology*, pages 6–1, 2006.

[46] D. Gusfield, *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, 1997.

[47] E. McCreight, "A space-economical suffix tree construction algorithm", *Journal of the ACM*, vol. 23, no. 2, pages 262–272, 1976.

[48] E. Ukkonen, "On-line construction of suffix trees", *Algorithmica*, vol. 14, no. 3, pages 249–260, 1995.

[49] Rainer Gerhards *et al.*, "The syslog protocol", RFC 5424, March, Tech. Rep., 2009.

[50] Tiobe. "Very long term history (sep. 2021)". (Sep. 2022), [Online]. Available: https://www.tiobe.com/tiobe-index/.

[51] Edson Barajas. "What is log management, and why is it important?" (Sep. 2022), [Online]. Available: https://www.tripwire.com/state-of-security/incident-detection/log-management-siem/log-management-why-important/.

[52] Risto Vaarandi, "Simple event correlator for real-time security log monitoring", *Hakin9 Magazine*, vol. 1, no. 6, pages 28–39, 2006.

[53] Microsoft. "Windows notepad". (Sep. 2022), [Online]. Available: https://apps.microsoft.com/store/detail/windows-notepad/9MSMLRH6LZF3?hl=pt-pt&gl=pt.

[54] I. Witten, E. Frank, M. Hall, and C. Pal, *Data mining: practical machine learning tools and techniques*, 4th. Morgan Kauffmann, 2016, ISBN: 9780128042915.

[55] W3C. "Extensible markup language (xml) 1.0 (fifth edition)". (Sep. 2022), [Online]. Available: https://www.w3.org/TR/xml/.

Requisites Specification

This appendix contains the requisites specification for the project in the form of Use Cases. In Section A.1, we define the use cases. Section A.2 contains the supplemental information necessary for the validation of inputs and expected constraints for the fields.

This software does not make distinction between the type of actors which interact with it. The Actors portion of the use cases will be omitted since it will always be the same: User.

A.1 Use cases

Table A.1 enumerates the use cases.

Table A.1: Use case table

Use Case	Name
UC-01	Log file analysis
UC-02	Filter on log file analysis
UC-03	Search on log file analysis
UC-04	Clear option in log file analysis
UC-05	Export option in log file analysis
UC-06	Save file
UC-07	Log file monitoring
UC-08	Parsing profiles management
UC-09	Parsing profiles creation
UC-10	Parsing profiles editing
UC-11	Parsing profiles deletion
UC-12	Metrics profiles management
UC-13	Metrics profiles creation
UC-14	Keyword profile
UC-15	Metrics profiles editing
UC-16	Metrics profiles deletion
UC-17	Organization

The following sections describe in detail the use cases found in Table A.1.

A.1.1 Log file analysis (Use case 1)

Summary: User uses the Analysis feature.

Preconditions: The user has a parsing profile and a metrics profile created.

Main Scenario:

1. The use case starts when a user chooses the “Analysis” tab in the main screen.
2. The system presents the user with the "File analysis setup" screen which has three inputs: “Choose file”, “Choose parsing profile”, “Choose metrics”, and a "Start" button, which is disabled.
3. The user presses “Choose file”.
4. The system opens an explorer for the user to select a log file.

5. The user selects a log file.
6. The system updates the display with the selected log file.
7. The user selects a parsing profile from the drop-down box.
8. The system updates the display with the selected profile.
9. The user selects a metrics profile from the drop-down box.
10. The system updates the display with the selected profile and activates the “Start” button.
11. The user presses “Start”.
12. The system launches a “File Analysis” screen, with the results.
13. The use case ends.

Alternative Scenario A

1. In step 2, the user chooses an invalid file.
2. The system gives an error popup and clears the selected file.
3. Resume from step 2.

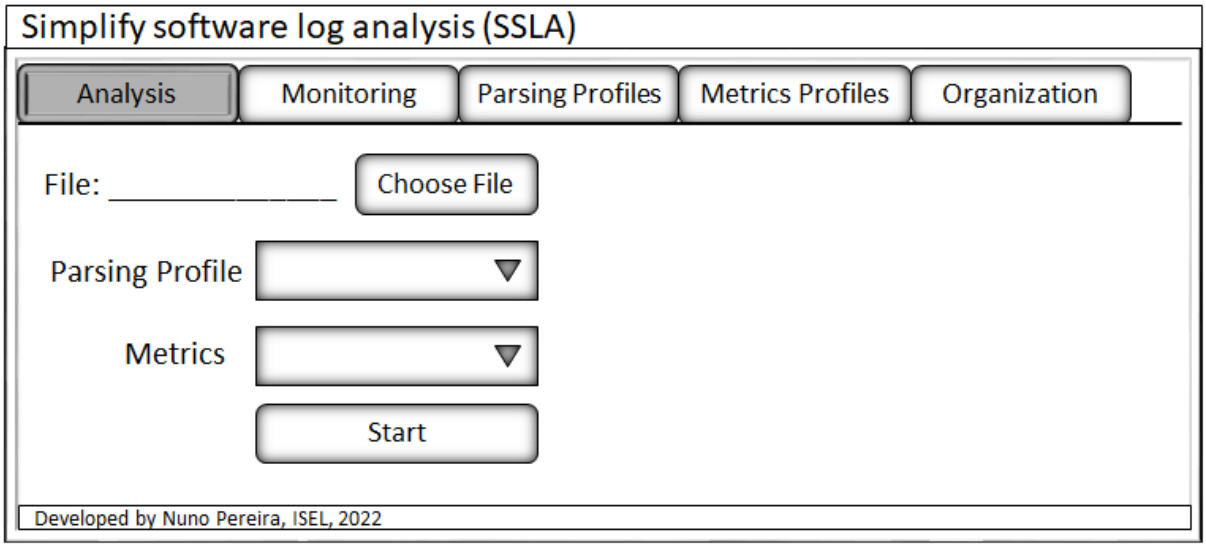


Figure A.1: File analysis setup screen.

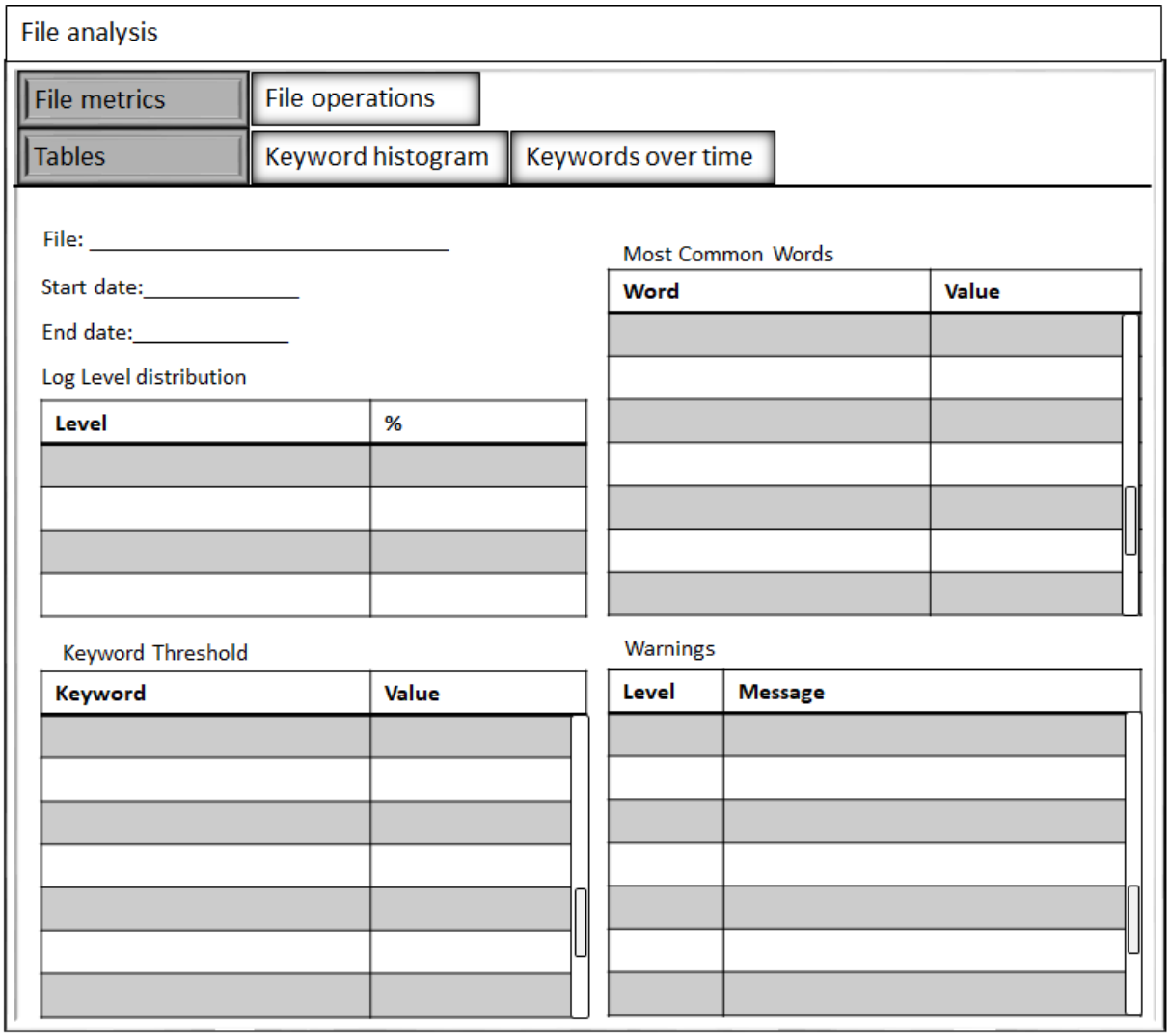


Figure A.2: File analysis metrics screen.

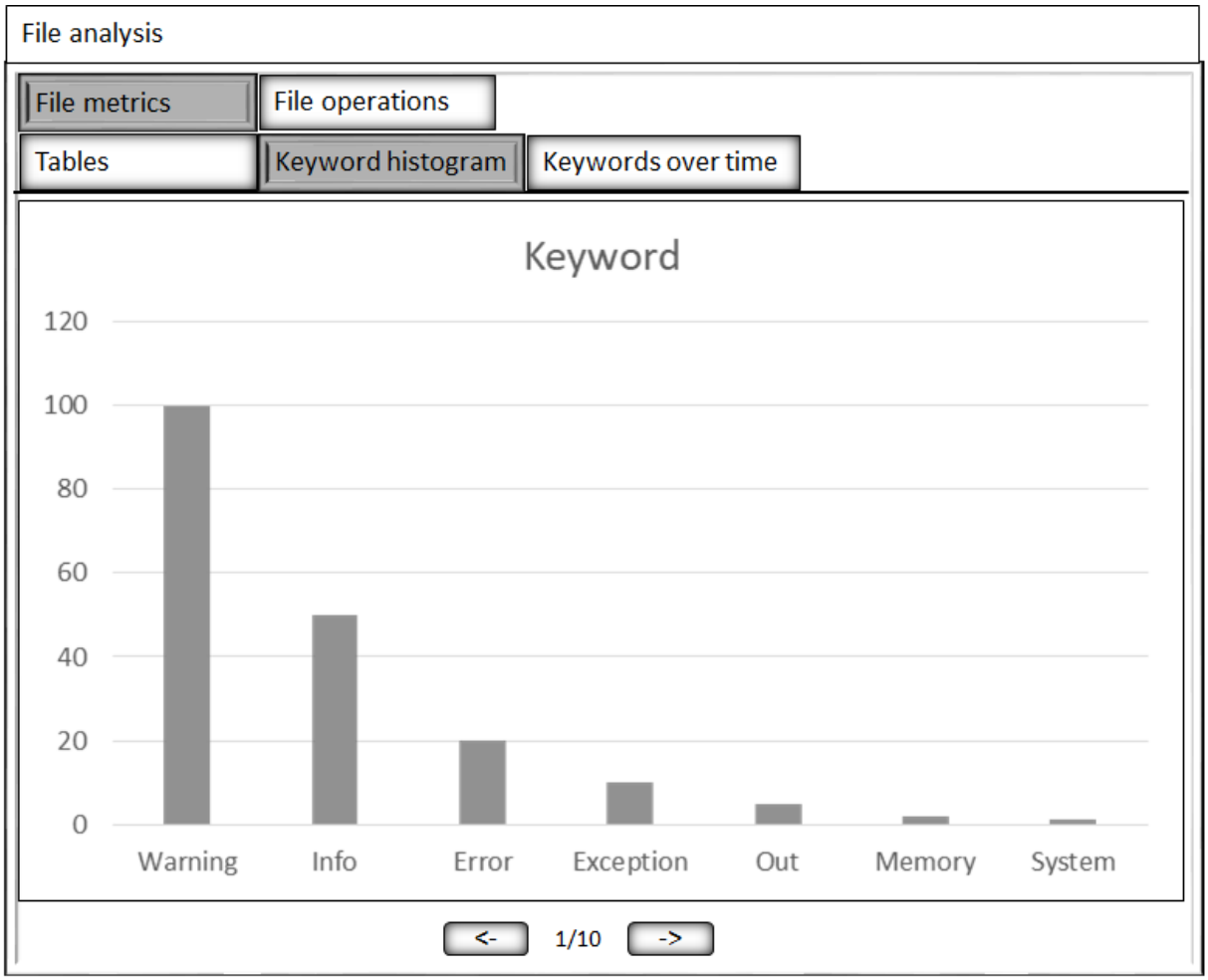


Figure A.3: File analysis metrics: keyword histogram screen.

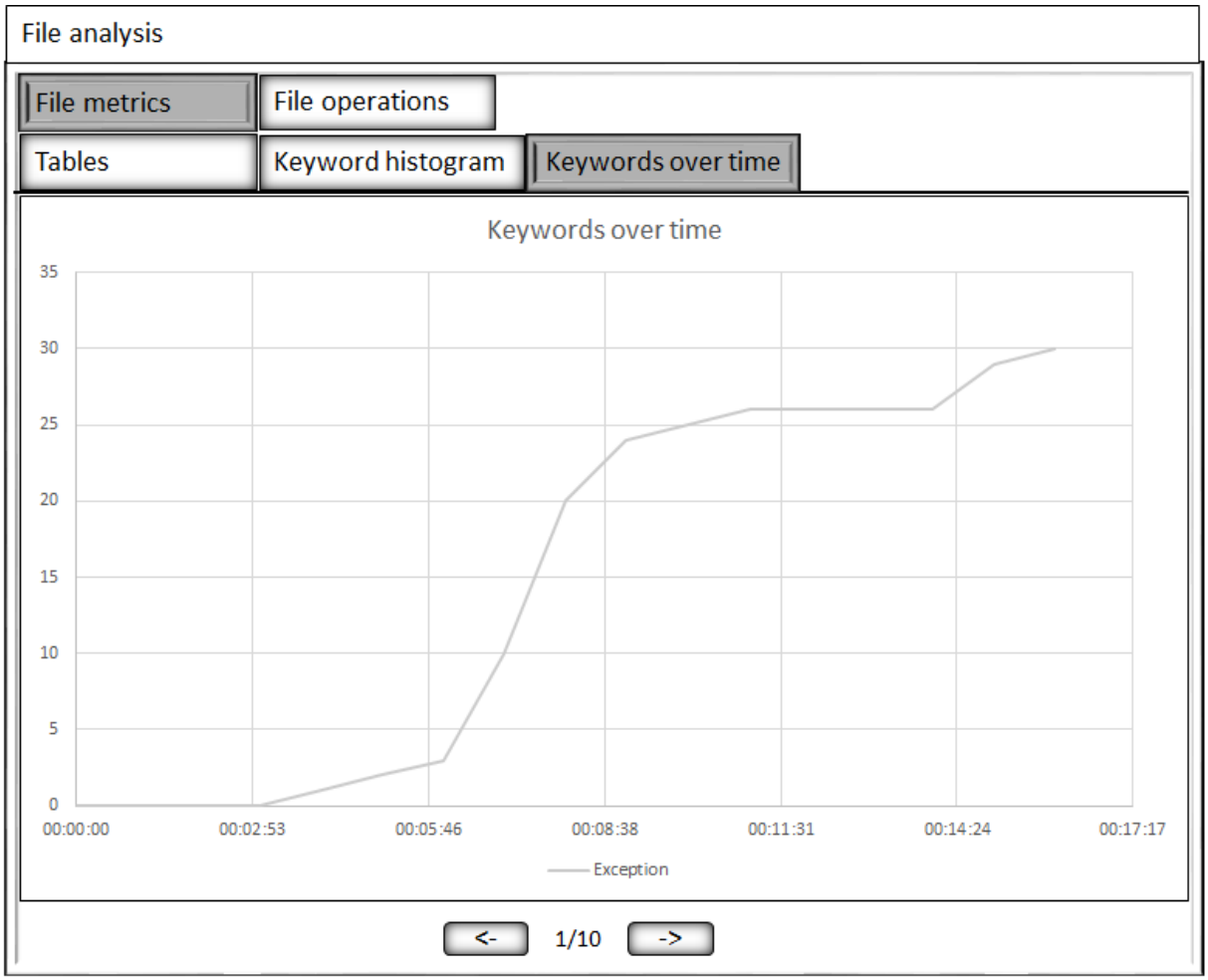


Figure A.4: File analysis metrics: keywords over time screen.

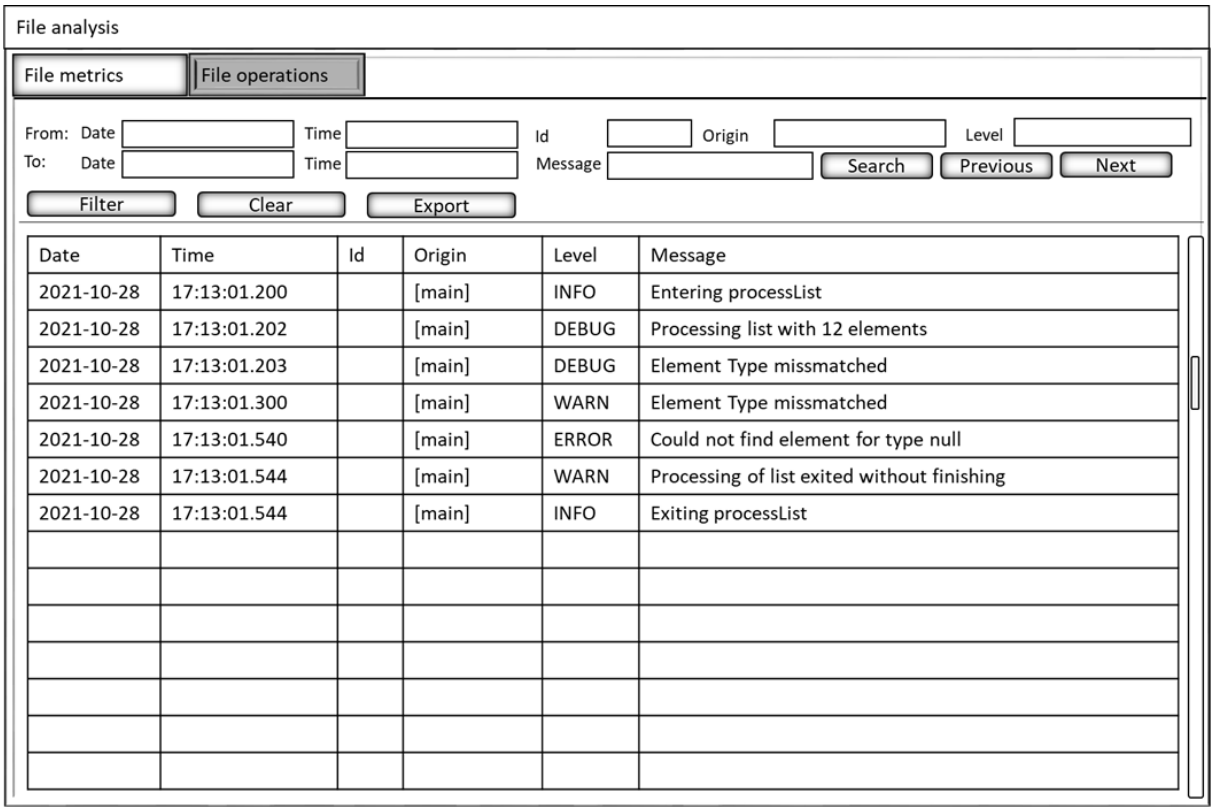


Figure A.5: File analysis details screen.

A.1.2 Filter on log file analysis (Use case 2)

Summary: User uses the Filter feature in the File Analysis screen.

Preconditions: The user has used the File Analysis feature.

Main Scenario:

1. The use case starts when the user fills the desired input fields and chooses the option “Filter” in the “File Analysis” screen.
2. The system validates the fields (according to Section A.2) and filters the contents based on those parameters. The contents on the table are updated accordingly.
3. The use case ends.

Alternative Scenario A

1. In step 2, one or more fields do not respect the requirements. The system marks the fields on screen.
2. The user corrects the fields and presses “Filter”.

3. The use case resumes in step 2.

Alternative Scenario B

1. In step 2, there is an issue processing the file.
2. The system notifies the user of the issue, and requests the user to validate the state of the file.
3. The use case ends.

A.1.3 Search on log file analysis (Use case 3)

Summary: User uses the Search feature in the File Analysis screen.

Preconditions: The user has used the File Analysis feature.

Main Scenario:

1. The use case starts when the user fills the "Message" input field and chooses the option "Search" in the "File Analysis" screen.
2. The system validates the field (according to Section A.2) and searches the contents based on those parameters. The contents on the table are updated accordingly.
3. The use case ends.

Alternative Scenario A

1. In step 2, the "Message" field does not respect the requirements. The system marks the field on the screen.
2. The user corrects the fields and presses "Search".
3. The use case resumes in step 2.

Alternative Scenario B

1. In step 2, there is an issue processing the file.
2. The system notifies the user of the issue, and requests the user to validate the state of the file.
3. The use case ends.

A.1.4 Clear option in log file analysis (Use case 4)

Summary: User uses the Clear feature in the File Analysis screen.

Preconditions: The user has used the File Analysis feature and used the Filter or Search feature.

Main Scenario:

1. The use case starts when the user presses the "Clear" button on the screen.
2. The system reloads the contents of the file from memory and clears all fields.
3. The use case ends.

A.1.5 Export option in log file analysis (Use case 5)

Summary: User uses the Export feature in the File Analysis screen.

Preconditions: The user has used the File Analysis feature.

Main Scenario:

1. The use case starts when the user presses the "Export" button on the screen.
2. The system opens a dialog and requests a name and a location for the file.
3. The user enters a name for the file and a location.
4. The system exports the contents of the table to a new file with the given name.
5. The use case ends.

Alternative Scenario A

1. There is an issue in step 4. The system notifies the user of the issue.
2. The use case ends.

A.1.6 Save file (Use case 6)

Summary: User uses a feature which results in a file being written to the computer.

Preconditions: The user has used the File Analysis feature.

Main Scenario:

1. The use case starts when the user uses an operation such that it writes a file.
2. The system opens a dialog and requests a name and a location for the file.
3. The user enters a name for the file and a location.
4. The system writes the file to the chosen location.
5. The use case ends.

Alternative Scenario A

1. In step 3, the indicated file already exists. The system requests confirmation from the user.
2. The user confirms writing over the file.
3. The use case resumes in step 4.

Alternative Scenario B

1. In step 3, the indicated file already exists. The system requests confirmation from the user.
2. The user chooses to not write over the file.
3. The use case resumes in step 2.

A.1.7 Log file monitoring (Use case 7)

Summary: User uses the Monitoring feature.

Preconditions: The user has an existing parsing profile and a metrics profile created.

Main Scenario:

1. The use case starts when a user chooses the “Monitoring” tab in the main screen.

2. The system presents the user with a new screen with three inputs: “Choose file”, “Choose parsing profile”, “Choose metrics”, and a "Start" button, which is disabled.

3. The user presses “Choose file”.

4. The system opens an explorer for the user to select a log file.

5. The user selects a log file.

6. The system updates the display with the selected log file.

7. The user selects a parsing profile from the drop-down box.

8. The system updates the display with the selected profile.

9. The user selects a metrics profile from the drop-down box.

10. The system updates the display with the selected profile and activates the “Start” button.

11. The user presses “Start”.

12. The system launches a “File Monitoring” screen, with the results. The tables are updated for each new line in the log file. If a threshold it surpassed or met, a warning is displayed to the user.

13. The user presses the “Stop” button.

14. The system stops processing the file.

15. The use case ends.

Alternative Scenario A

1. In step 2, the user chooses an invalid file.

2. The system gives an error popup and clears the selected file.

3. Resume from step 2.

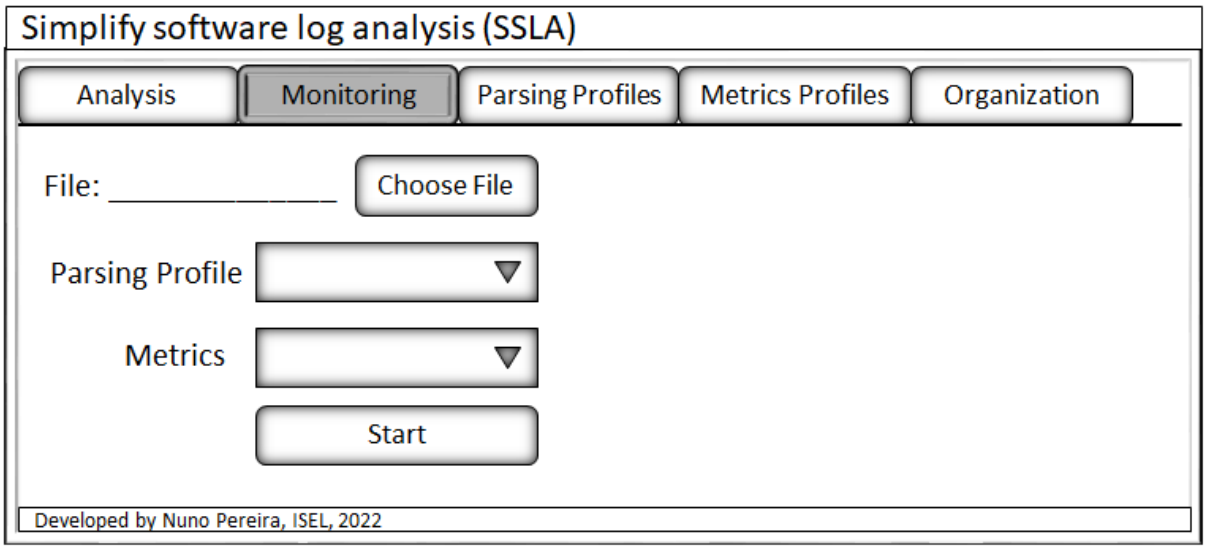


Figure A.6: File monitoring setup screen.

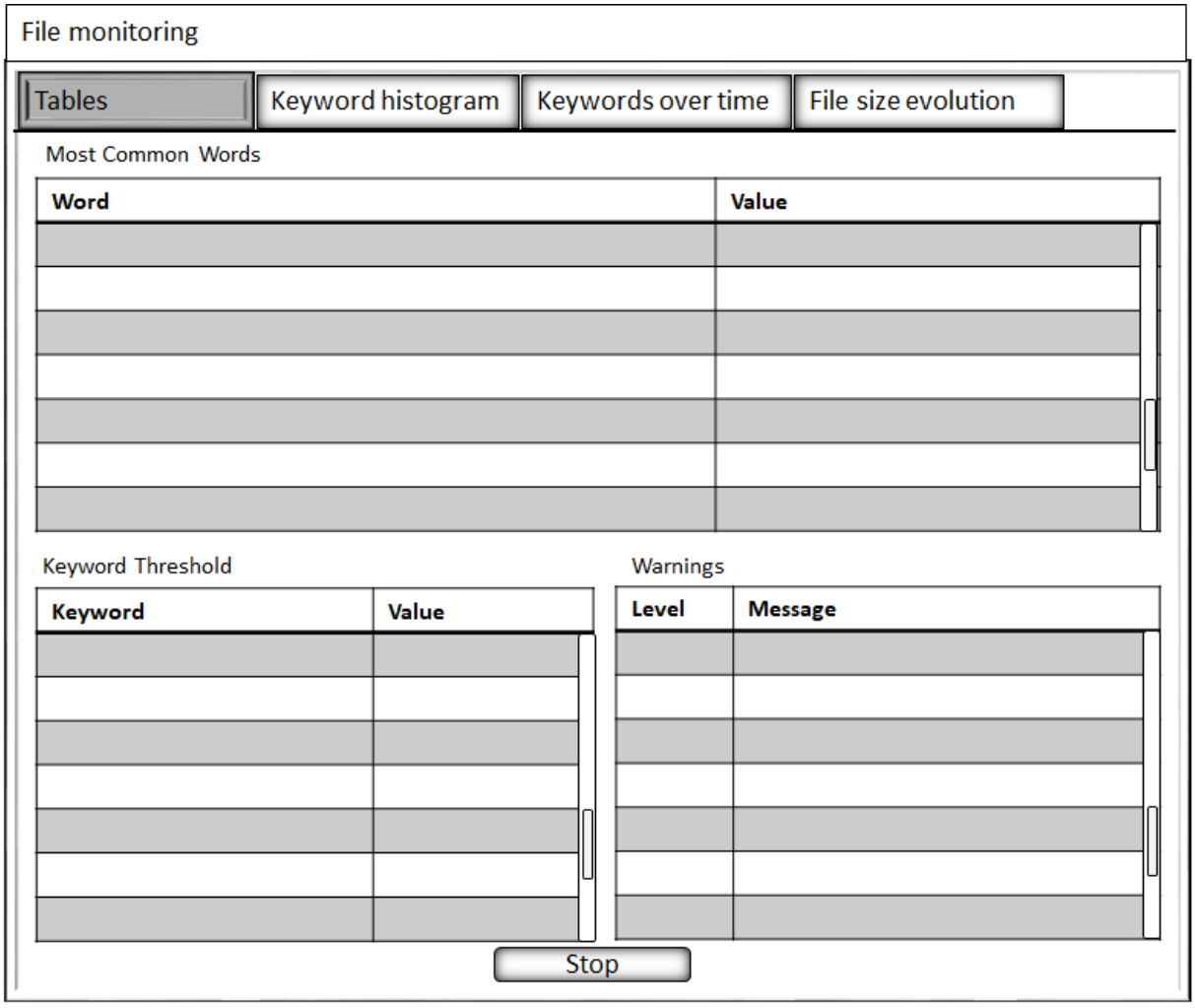


Figure A.7: File monitoring metrics screen.

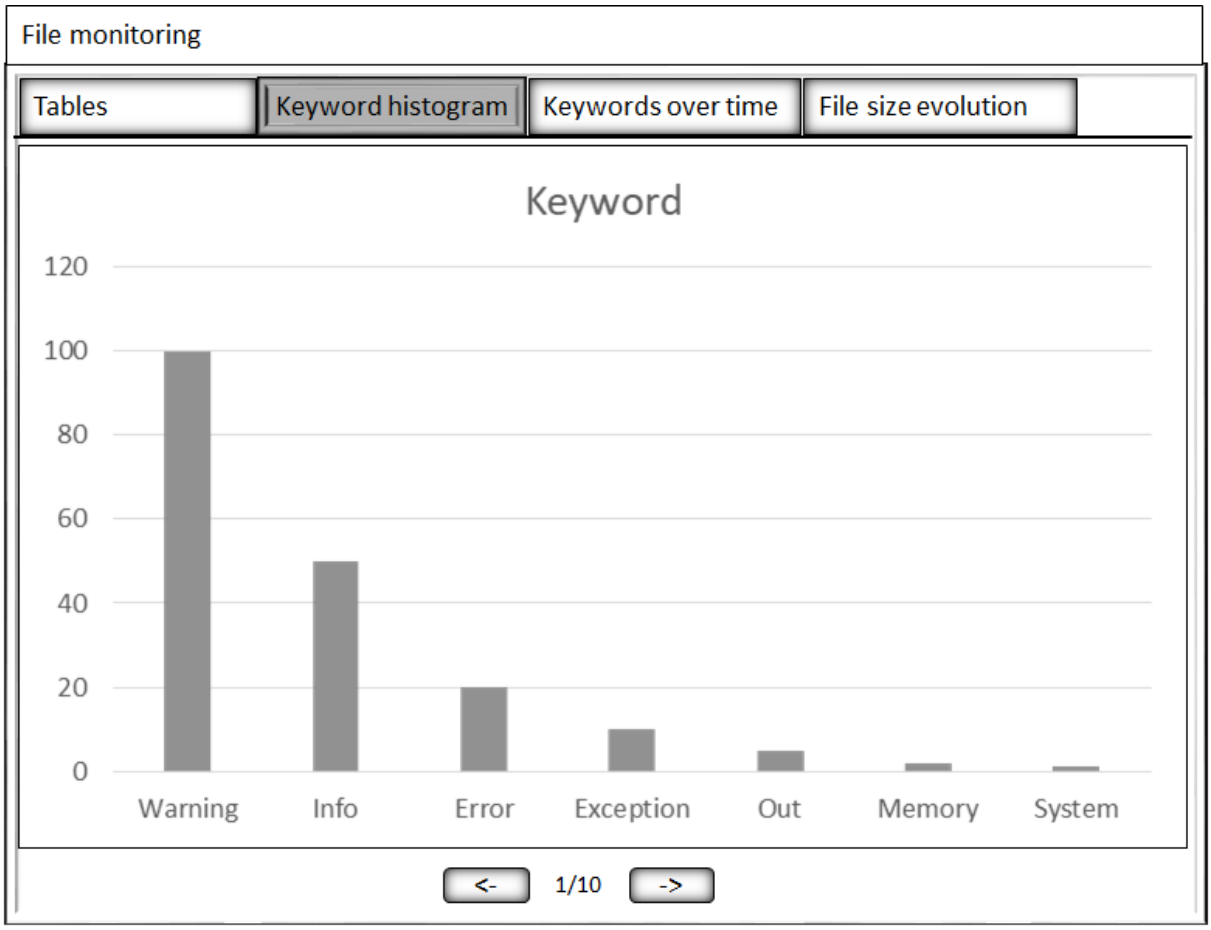


Figure A.8: File monitoring metrics: keyword histogram screen.

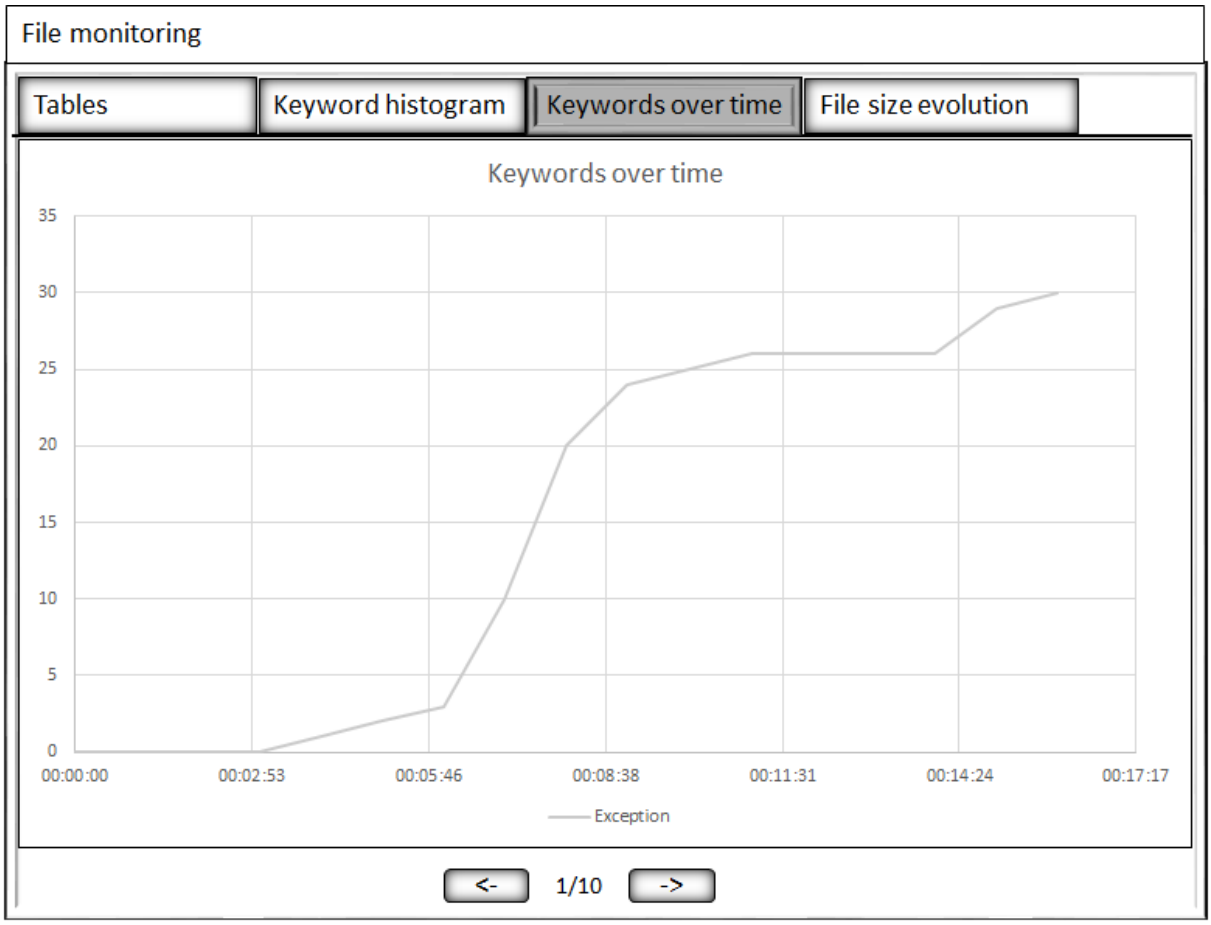


Figure A.9: File monitoring metrics: keyword over time screen.

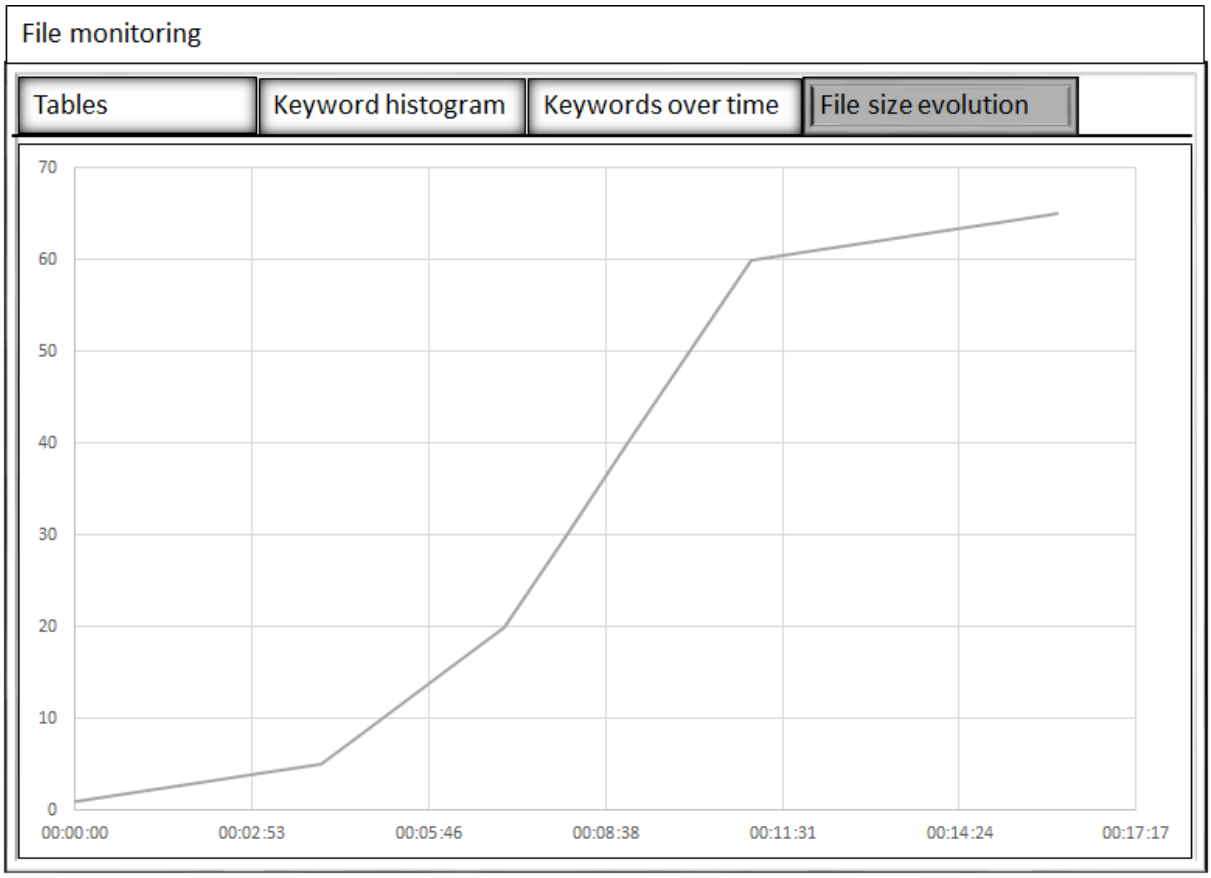


Figure A.10: File monitoring metrics: file size evolution screen.

A.1.8 Parsing profiles management (Use case 8)

Summary: User uses the Parsing Profiles management feature.

Preconditions: None.

Main Scenario:

1. The use case starts when a user chooses the “Parsing Profiles” tab in the main screen.

2. The system opens the "Parsing Profile Management" screen listing the existing parsing profiles.

3. The use case ends.

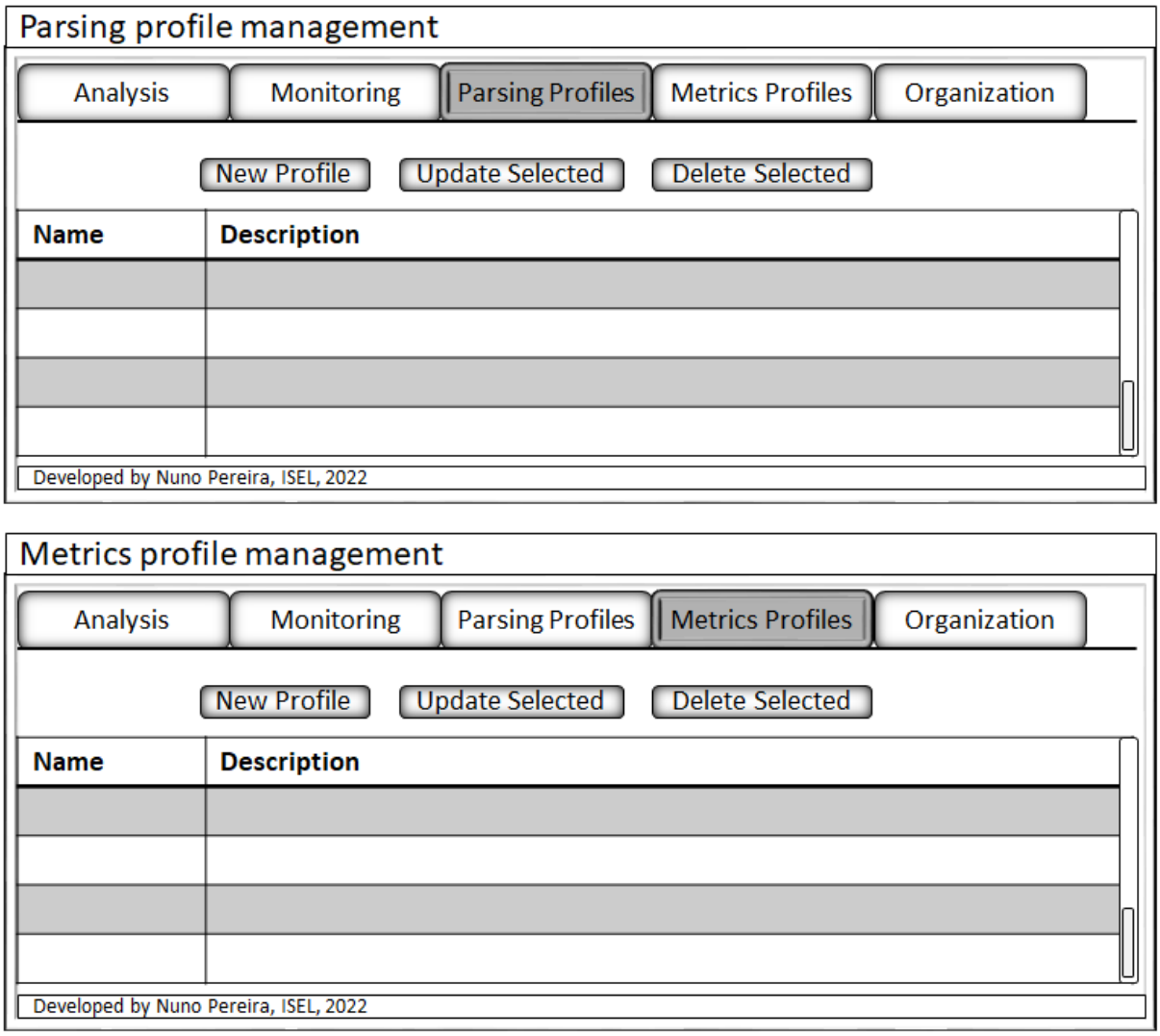


Figure A.11: Parsing profile management and metrics profile management screens.

A.1.9 Parsing profiles creation (Use case 9)

Summary: User creates a parsing profile.

Preconditions: The user has used the Parsing Profiles feature

Main Scenario:

1. The use case starts when a user chooses the "New Profile" tab in the "Parsing Profiles" screen.
2. The system opens the "Parsing Profile Editor" screen.
3. The user gives a name to the profile.

4. The system validates the field name.

5. The user chooses a text class, indicates if the class is meant to be ignored upon parsing and if there is a specific format for the text class, and presses "Add".

6. The system adds the Text Class to the "Result" field. The system enables the Separator fields and disables the Text Class fields.

7. The user chooses a Separator. The system enables the Text Class fields and disables the Separator fields.

8. The steps 6 and 7 may be repeated as needed.

9. The user presses the "Save Profile" button.

10. **Include save file use case, described in Section A.1.6**.

11. The use case ends.

Alternative Scenario A

1. The user presses the "Remove Last" button.

2. The system removes the last entry in the "Result" field, and enables the corresponding option Text Class or Separator.

Alternative Scenario B

1. The user presses the "Clear" button.

2. The system removes all the entries in the "Result" field, and enables the Text Class options.

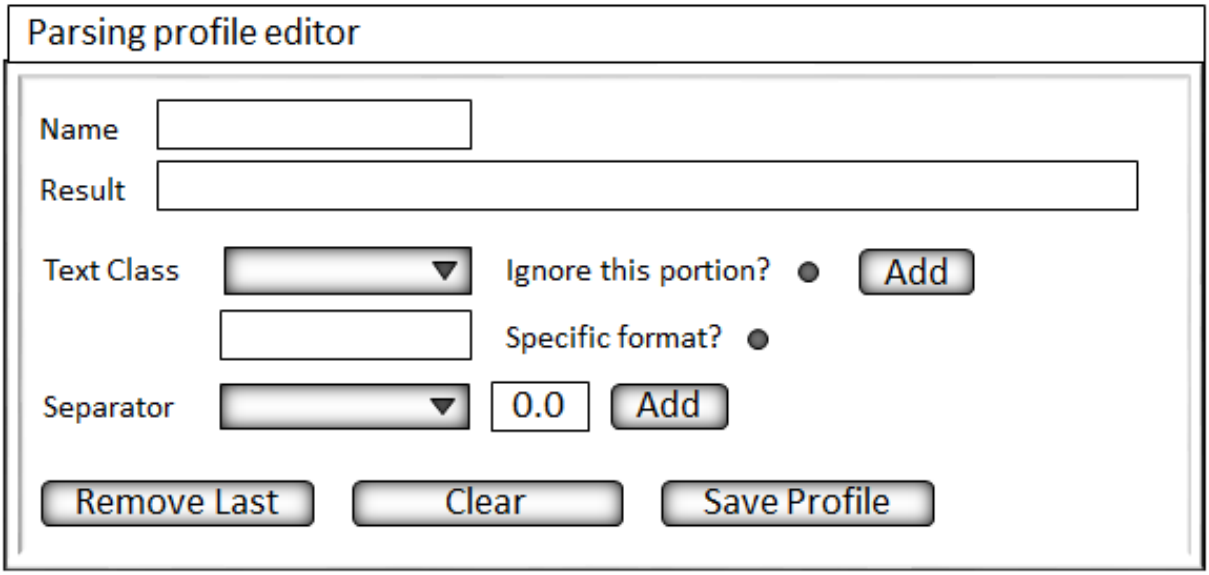


Figure A.12: Parsing profile editor screen.

A.1.10 Parsing profiles editing (Use case 10)

Summary: User edits an existing parsing profile.

Preconditions: The user has used the Parsing Profiles feature and there is one parsing profile created.

Main Scenario:

1. The use case starts when a user selects a line in the profiles table.
2. The system enables the "Update Selected" option.
3. The user presses the "Update Selected" button.
4. The system opens the "Parsing Profile Editor" screen.
5. The user changes the desired values, following the behavior in use case 8, present in Section A.1.8.
6. The user presses the "Save Profile" button.
7. **Include save file use case, described in Section A.1.6**.
8. The use case ends.

A.1.11 Parsing profiles deletion (Use case 11)

Summary: User deletes an existing parsing profile.

Preconditions: The user has used the Parsing Profiles feature and there is one existing parsing profile.

Main Scenario:

1. The use case starts when a user selects a line in the profiles table.
2. The system enables the "Delete Selected" option.
3. The user presses the "Delete Selected" button.
4. The system requests confirmation from the user.
5. The user confirms the deletion.
6. The system disables the "Delete Selected" button and refreshes the table.
7. The use case ends.

A.1.12 Metrics profiles management (Use case 12)

Summary: User uses the Metrics Profiles management feature.

Preconditions: None.

Main Scenario:

1. The use case starts when a user chooses the "Metrics Profiles" tab in the main screen.
2. The system opens the "Metrics Profile Management" screen listing existing metrics profiles.
3. The use case ends.

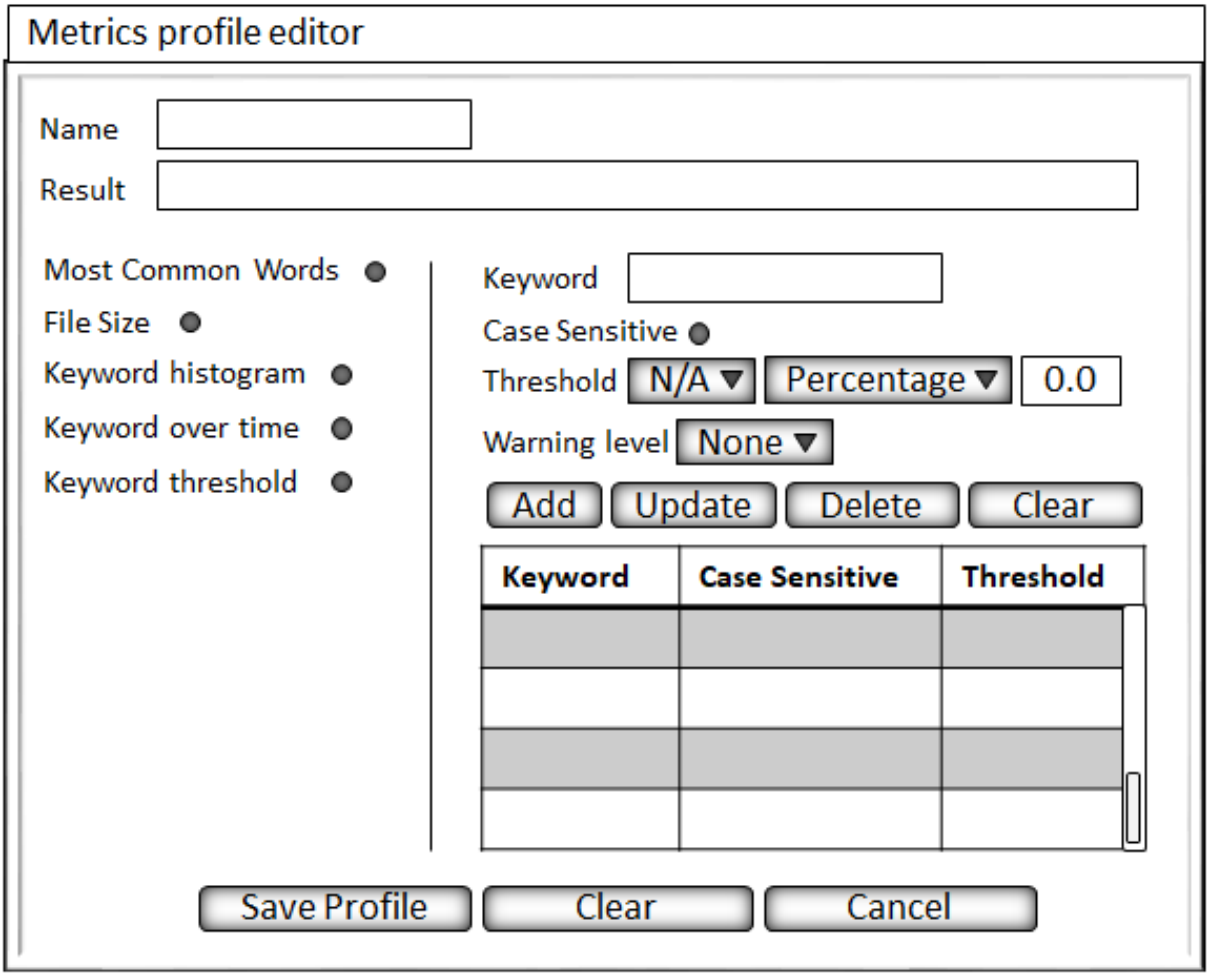


Figure A.13: Metrics profile editor screen.

A.1.13 Metrics profiles creation (Use case 13)

Summary: User creates a metrics profile.

Preconditions: The user has used the Metrics Profiles feature.

Main Scenario:

1. The use case starts when a user chooses the option "New Profile" in the "Metrics Profiles" screen.
2. The system opens the "Metrics Profile Editor" screen.
3. The user gives a name to the profile.
4. The system validates the field name.
5. The user chooses the metrics to be considered.

6. The system updates the "Result" field accordingly.
7. **Include keyword profile use case, described in Section A.1.14**.
8. The user presses the "Save Profile" button.
9. **Include save file use case, described in Section A.1.6**.
10. The use case ends.

Alternative Scenario A

1. The user deselects any Metric.
2. The system removes the entry in the "Result" field.

A.1.14 Keyword profile (Use case 14)

Summary: User edits the keywords on a metrics profile.

Preconditions: The user has used the Metrics Profiles feature.

Main Scenario:

1. The use case starts when a user presses the "Add" button after adding a keyword in the "keyword" input field, selects if the word is meant to be processed as case-sensitive or not and optionally defines a threshold.
2. The system adds a new line to the table with the given values.
3. The user selects the line in the table.
4. The system enables the "Update" and "Delete" buttons and updates the fields with the values.
5. The user edits the values and presses the "Update" button.
6. The system updates the line.
7. The user selects another line.
8. The system enables the "Update" and "Delete" buttons and updates the fields with the values.
9. The user presses the "Delete" button.
10. The system removes the line from the table.
11. The use case ends.

A.1.15 Metrics profiles editing (Use case 15)

Summary: User edits an existing metrics profile.

Preconditions: The user has used the Metrics Profiles feature and there is one metrics profile created.

Main Scenario:

1. The use case starts when a user selects a line in the profiles table.
2. The system enables the “Update Selected” option.
3. The user presses the “Update Selected” button.
4. The system opens the “Metrics Profile Editor” screen.
5. The user changes the desired values, following the behavior in use case 13, described in Section A.1.13.
6. The user presses the “Save Profile” button.
7. **Include save file use case, described in Section A.1.6**.
8. The use case ends.

A.1.16 Metrics profiles deletion (Use case 16)

Summary: User deletes an existing metrics profile.

Preconditions: The user has used the Metrics Profiles feature and there is one metrics profile created.

Main Scenario:

1. The use case starts when a user selects a line in the profiles table.
2. The system enables the “Delete Selected” option.
3. The user presses the “Delete Selected” button.
4. The system requests confirmation from the user.
5. The user presses "Ok" confirming the deletion.
6. The system disables the “Delete Selected” button and refreshes the table.
7. The use case ends.

A.1.17 Organization (Use case 17)

Summary: User uses the organization feature.

Preconditions: The user has used the Metrics Profiles feature and there is one metrics profile created.

Main Scenario:

1. The use case starts when the user chooses the tab "Organization" in the main screen.
2. The system presents the user with the "Organization screen".
3. The user presses the "Choose" button next to the source folder.
4. The system presents an explorer for the user to choose a folder.
5. The user chooses a folder.
6. The system fills the name of the folder in the screen and updates the table with the contents of the folder.
7. The user presses the "Choose" button next to the target folder.
8. The system presents an explorer for the user to choose a folder.
9. The user chooses a folder.
10. The system fills the name of the folder in the screen and updates the table with the contents of the folder.
11. The user chooses the parsing profile, the metrics profile, and the files that shall be analysed. The user presses the "Copy" or "Move" buttons.
12. The system parses and analyses the selected files based on the settings. Applying the operation chosen by the user, in which all the thresholds are surpassed or met.
13. The use case ends.

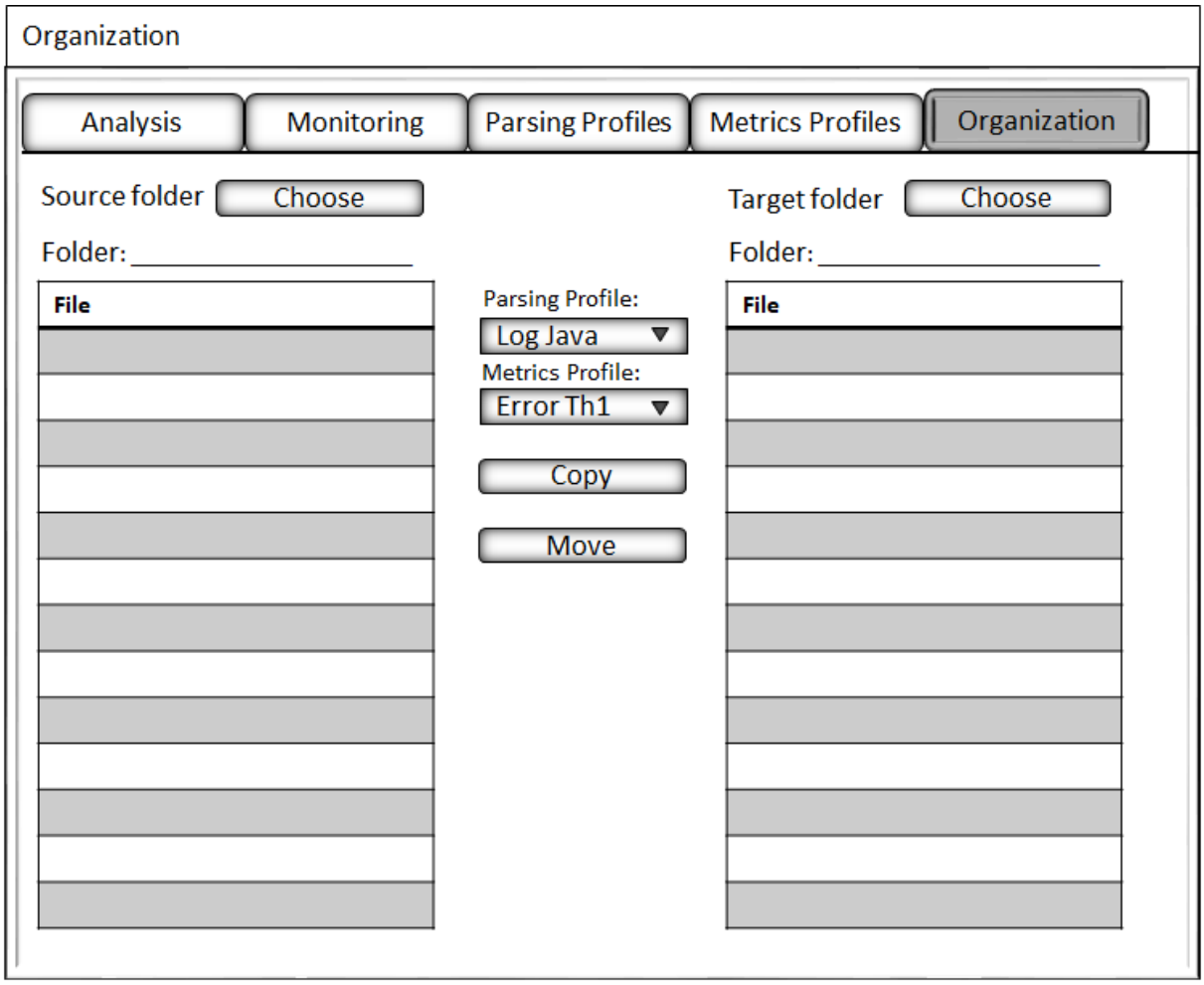


Figure A.14: File organization screen.

A.2 Supplementary specification constraints

1. File names limited to 100 characters.
2. Keywords limited to 50 characters.
3. Threshold as percentage: floating point, two decimal places, range 0 to 1.
4. Threshold as number of occurrences: integer, range 1 to 1000.
5. Specific format field limited to 20 characters.
6. Method limited to 100 characters.
7. Level limited to 10 characters.
8. Message limited to 255 characters.

9. Date: date.
10. Time: time.
11. Profile names limited to 30 characters. Profile names may not repeat withing the same category.
12. Threshold type: N/A, >, >=, =, <=, <.
13. Threshold unit: Occ (Occurrences), %, None.

B

Additional Experimental Results

B.0.1 Impact on search times

Table B.1 shows the full data used to create the summary present in Table 5.5.

For this test we used a file with an average expected message size of 100 characters with 100MB and over 750k lines.

We tested three search algorithms, one without a support structure, which reads the string using indexOf() counting the located indexes, and two algorithms based in Suffix Arrays. For the Suffix Array algorithms we used an iterative and a recursive implementation of the binary search. We ran the same test twenty times for each algorithm.

The results showed that the search made without a search structure is consistently quicker than using a support structure like a Suffix Array, and that the iterative implementation is faster than the recursive one. This can be explained by the overhead caused by the repeated function calls [21]. Besides possible mistakes in our implementation of the algorithm, what we conclude is that already existing search algorithms in Java are optimized the best they can be.

Table B.1: Search time results 100 MB size file with average message size

Without Support Structure	Suffix Array Iterative Search	Suffix Array Recursive Search
Time in milliseconds	Time in milliseconds	Time in milliseconds
96,0633	318,5794	286,0373
78,7637	248,6042	239,4381
29,5918	237,9351	273,5563
115,6485	204,6567	269,2998
60,0245	226,3297	254,221
112,1589	196,2367	256,9929
53,5096	241,9007	259,9924
96,9529	234,7066	260,2429
53,5395	240,7573	267,7088
105,4748	201,2981	260,3404
54,963	220,0367	232,8997
117,2548	232,693	273,446
52,6133	217,6281	254,3001
114,1473	227,4337	242,8685
54,3976	229,9987	279,7106
74,2793	259,8736	251,1946
82,1498	179,1487	279,1559
55,0653	237,1474	279,2554
81,9351	239,9645	273,0364
110,3345	280,5181	217,5353
Average time	Average time	Average time
79,943375	233,77235	260,56162