Classification and Analysis of Inner Source Development Artifacts

MASTER THESIS

Benedict Martin Weichselbaum

Submitted on 2 November 2023

Friedrich-Alexander-Universität Erlangen-Nürnberg
Faculty of Engineering, Department Computer Science
Professorship for Open Source Software

Supervisor:
Stefan Buchner, M.Sc.
Prof. Dr. Dirk Riehle, M.B.A.

Friedrich-Alexander-Universität
Faculty of Engineering

Declaration of Originality

I confirm that the submitted thesis is original work and was written by me without further assistance. Appropriate credit has been given where reference has been made to the work of others. The thesis was not examined before, nor has it been published. The submitted electronic version of the thesis matches the printed version.

Erlangen, 2 November 2023

License

Erlangen, 2 November 2023

Abstract

Inner source software development describes the usage of open-source development practices within organizations. To make this process more transparent, classification of the software development artifacts is needed. These classifications can then be used for further analysis. To create a classification system, the design science methodology was used as a structure for the thesis. The key research questions were what objectives the classification system had to fulfill, how such a system can be designed and implemented, and what kind of analytics are possible with such a system. The objectives stated that the classification system has to be able to process different kinds of text-based software artifacts, take these artifacts from various data sources, create usable classifications for analytics, and do all of this without using machine learning techniques because these do not provide reproducibility when using different training datasets. For the design, a data pipeline is defined that extracts, preprocesses, classifies, post-processes, and writes the software artifact data. A non-complete set of classifications for different artifact categories is defined and designed. For the implementation, a Python program gets conceptualized, allowing for the demonstration of the original design. With the classifications at hand, example analytics are proposed to show the applicability of the results. The final evaluation shows that the set objectives were fulfilled by the design of the software artifact classification system.

Contents

List of Figures

List of Tables

Listings

Acronyms

API Application Programming Interface

CLI Command Line Interface

ETL Extract Transform Load

JSON JavaScript Object Noation

JVM Java Virtual Machine

MECE Mutually Exclusive, Collectively Exhaustive

URL Uniform Resource Locator

1 Introduction

1.1 Motivation

The term "inner source" describes the usage of open-source development practices within companies. These organizations may still develop proprietary software but internalize an open source-like culture within their inner processes by opening up the internal development of software. Some key characteristics for inner source development are open communication, open development artifacts, communities around software, and re-use characteristics. (Capraro and Riehle, 2016)

Although the application of inner source within organizations can be beneficial, the open nature of inner source may cause some management, accounting, and taxation problems. For example, because code is no longer developed within strict organizational boundaries, accounting issues can become a concern.

To counteract these problems, precise measurement and assessment of the software development process is needed within organizations that work with inner source principles. Currently, is it possible to monitor the overall development artifacts like source code, commit messages, e-mails, and instant messages. However, to comprehensively understand the development processes within an organization, more information about these artifacts is needed. To gain more information, the mentioned artifacts have to be classified. Possible classifications or tags could be the differentiation of productive code and comments within source code or the breakdown of e-mail traffic into classes like "original e-mail" or "forward e-mail".

This thesis looks closely into the stated problem by creating a classification algorithm for software development artifacts to enable better analysis in the context of inner source software development. To achieve this goal, the issue, together with related tools and literature, gets analyzed to create a software design that can classify different software artifacts. An exemplary implementation and demonstration will show the applicability of the solution. Based on the created classifications, different analysis suggestions will be shown to demonstrate the value such a classification system yields.

1.2 Research Questions and Structure

From the motivation, different research questions can be formulated that the thesis answers. They also provide a guideline for the whole work, creating logical steps in creating the classification system. The thesis itself is structured after the design science methodology described by Pfeffers et al. (Peffers et al., 2007)

1. **What are the objectives for a software development artifact classification system?**

The first research question aims at analyzing the classification problem at hand by defining the objectives for the software. Only with clear objectives a coherent software design later is possible. Moreover, when demonstrating and evaluating the exemplary implementation, the objectives are needed to see if the implementation is sufficient. For answering the first research question, the problem identification (Chapter 2) and objective definition (Chapter 3) parts of the design science method are used. Furthermore, related work within the problem identification will show why the stated problem has not yet been solved satisfactorily.

2. **How can such a classification system be conceptually designed and implemented?**

After the objectives are clear, a solution design and implementation is needed. The solution designed here will be designed in a way that is generalized to support potentially many kinds of software artifacts. The implementation will look at an example system with a finite set of artifacts and data sources. These sections of the thesis cover the solution design (Chapter 4) and demonstration (Chapter 5) parts of the design science methodology. As stated before, an evaluation (Chapter 7) based on the objectives will also be conducted.

3. **What kinds of analysis are possible with the developed classification system?**

With an exemplary classification system implemented, the last question remaining is how such a system benefits organizations which apply the inner source methodology. For that, different analysis based on the classifications are needed to gain value. This thesis will provide several examples for such analytics.

2 Problem Identification

As described in the previous section, the goal of this thesis is to create a software artifact classification system. Therefore, this section will provide a detailed problem identification and take related work into account.

2.1 Related Work and Tools

In the first part of the problem identification related work and tools will be taken into account. The related work will help to show the need for a general software artifact classification system by demarcating it from existing classification ideas. The related tools show approaches for further investigation when designing concrete classifiers.

2.1.1 Text-based Classification for Software Artifacts

Text-based classification for software artifacts is mentioned in various papers trying to fulfill different goals. Alqahtani and Rilling, for example, describe in their paper how they used artifacts like code, emails, and bug reports to tag them for security purposes. For their endeavor, they propose an algorithm that parses, tokenizes, and cleans the text-based data. The resulting "bag of words" is used to map terms to certain security topics. (Alqahtani and Rilling, 2017)

Very similar to this thesis' approach, Ma et al., in their paper "Automatic Classification of Software Artifacts in Open-Source Applications" look into tagging open-source software artifacts. To accomplish that, different solutions are mentioned. Heuristic applications look at simple characteristics like filenames and file extensions to tag an artifact. Machine-learning algorithms are also suggested. Interestingly, as part of their paper, the authors also define the different kinds of software artifacts they classify. For non-documentation files, a file-extension heuristic is used. For documentation files, the tags are "contributor's guide", "design document", "license", "list of contributor", "release note", "requirement document" and "setup file". (Ma et al., 2018)

Another viewpoint on the topic is taken by Rani et al. In their paper, they only look at source code comments in different languages and classify them by comment type. For that, machine learning and natural language processing techniques are used. In the end, the authors present a classification algorithm for comments, sorting the artifacts into categories like "summary", "rationale" and "example", explaining what a source code comment is about. (Rani et al., 2021)

Similarly, Hindle et al. use commit logs and messages for the categorization or classification of large changes to the software repository. The acknowledges that many commits can be easily classified by the commit message and metadata. But for their work, they extend that process by introducing machine learning algorithms to categorize the changes into the categories "Corrective", "Adaptive", "Perfective", "Feature Addition" and "Non Functional". (Hindle et al., 2009)

A very comprehensive overview over the topic of software data analysis in general is given in the book "The Art and Science of Analyzing Software Data" by Bird et al. Especially chapter 3, "Analyzing Text in Software Projects" goes into detail on how to classify or "code" text artifacts in software projects. They also mention that analyzing textual artifact data is a field with unused potential. For the textual analysis, the book also defines different software artifacts based on the V-Model that are relevant. Besides that, tools for analysis are also part of the content. (Bird et al., 2015)

Another paper from Nazar et al. covers the related topic of software artifact summarization. In their literature review, they cover what kinds of software artifacts exist and what kinds of approaches exist for summarization, meaning receiving a reduced set of information about an artifact. (Nazar et al., 2016)

Other interesting approaches for classifying software artifacts come from Bacchelli et al. that categorize development e-mails into classes like "junk", "code" and "patch" with the help of machine learning. (Bacchelli et al., 2012) Also using machine learning, Yusof and Rana classify source code by using structural information like code metrics to improve the reuse and maintenance of these code artifacts. (Yusof and Rana, 2010)

2.1.2 Inner Source

Because this thesis has the goal of improving inner source development, relevant works should be mentioned. The term "inner source" was first coined by Tim O'Reilly. The definition of the term used in this thesis is taken from Riehle and Capraro who defined inner source by conducting a systematic literature review to find the elements that constitute inner source. They also point out the advantages and disadvantages of inner source. (Capraro and Riehle, 2016) Similar works were also done by Edison et al. who also did a literature review of inner source

publications. Their goal was to find the state-of-the-art in inner source research. (Edison et al., 2018)

Besides researching the inner source on a fundamental level, there are also works discussing the topic in a more practical way. The article "Inner Source - Adopting Open Source Development Practices in Organizations: A Tutorial" from Fitzgerald and Stol shows factors and ways on how to effectively implement the inner source philosophy within an organization. (Stol and Fitzgerald, 2015) Similarly, Stol et al. look into the real challenges that come with inner source development. They also mention certain legal problems that can occur with the method. (Stol et al., 2011)

Similar to this thesis' goal of making the inner source development process more transparent, Buchner and Riehle write in their conference paper "Calculating the Costs of Inner Source Collaboration by Computing the Time Worked" about time measurement in inner source development. This measurement is necessary for fiscal and administrative reasons, and therefore prompts a related problem to that of this thesis. (Buchner and Riehle, 2022)

2.1.3 Differentiation to Machine Learning

As already mentioned in section 2.1 many classification attempts for software development artifacts are based on machine learning techniques. Although such approaches are interesting for research and potentially real-world systems, they are not suitable in the context of this thesis. The motivation (section 1.1) stated that a more in-depth understanding of the development process is needed because Inner Source poses different legal difficulties. As a consequence, the insight into the development artifacts has to be precise and, more importantly, reproducible. Machine learning algorithms do not fit these criteria because the outcome depends on the training dataset. Therefore, this thesis will solve the classification problem without using machine learning algorithms.

2.1.4 Related Tools

Besides topic-related written content, there are also related software tools addressing software development artifact classification. Chaturvedi et al. did a comprehensive literature review of papers published in the proceedings of the conferences on mining software repositories. The paper categorizes the different tools based on their data processing. One of these categories is explicitly "Classification". The category contains tools for bug severity tagging, code fault detection, code clone detection, etc. Many of the tools again rely on machine learning, which is unfit for this thesis' approach to classification. (Chaturvedi et al., 2013)

Other than the tools mentioned in the paper, different tools exist that can be used directly or indirectly for artifact classification. If it is wanted to classify source code into comments and actual code, a tool like "cloc" (https://github.com/AlDanial/cloc) can help to achieve it. "cloc" counts the number of actual lines of code and comments in a file for various programming languages. In contrast to such straightforward tools, things like certain parsers or lexers could also help with certain classifications. For example, if a text file contains certain Markdown key symbols, they can be detected via a Markdown parser or lexer like "Marked" (https://github.com/markedjs/marked). The same would also be possible for source code. For example, by using a parser or syntax highlighter. Yet another tool that is usable for classification are log parsers like the "Jenkins" "Log Parser" plugin (https://plugins.jenkins.io/log-parser/). With a tool like this, it is possible to take a software build artifact and classify if it contains any errors, warnings, or other information.

All the examples given of related or usable tools for artifact classification are not sufficient for the desired classification system. Either they use machine learning techniques that are not wanted as part of the solution, or they only provide a partial solution to a subset of wanted classifications (see, e.g., "cloc"). These tools can be part of the final solution, but they do not provide all the processes necessary for a comprehensive classification system.

2.2 Classification System

With the related work and tools and a clear differentiation from machine learning in mind, the issue of the wanted classification itself can be looked into. For that, it has to be identified what kind of software artifacts need analyzing, what kind of data sources will be used for the classification system, and what the resulting classes will be for the various artifacts.

2.2.1 Software Development Artifacts

Before it can be defined what the classification system is supposed to do, it is necessary to point out what kind of software development artifacts are relevant. Although the term "software" is widely used, it is not necessarily clear what constitutes software. But knowing exactly what software is, is important because, for the classification system, it has to be known what should be analyzed. Pfeiffer (2020) in his article "What Constitutes Software? An Empirical, Descriptive Study of Artifacts" tries to answer this exact question. An essential point from his article is the fact that software is more than just source code. Two other high-level categories of software artifacts are mentioned: data and documentation. Data hereby means files like images, configurations, videos, etc. (Pfeiffer, 2020)

In the already mentioned literature review by Nazar et al. a list of relevant software development artifacts is given. (Nazar et al., 2016) The not comprehensive list contains:

- Bug Reports
- Source Code
- Mailing Lists

Another paper concerning software repository mining also lists different kinds of software development artifacts. Baysal et al. identify in their paper source code, version control metadata, defect tracking data, and electronic communication as software development artifacts. (Baysal et al., 2012)

In addition to that, Helmut Balzert in his book "Lehrbuch der Softwaretechnik - Basiskonzepte und Requiements Engineering" describes the requirements engineering process and explicitly mentions "requirements specifications" as a software artifact. These requirements specify the product from the customer's perspective. These specifications are often used in an agile development process in the form of user stories, which are the basis for individual tickets or issues within the software project. (Balzert, 2009)

To summarize these findings for the problem of software artifact classification, there is a clear set of artifacts that need to be analyzed. First and foremost, source code is one of the central and most important artifacts in every software development endeavor. It is critical to mention that source code does not only consist of productive statements, but also comments and test code. Furthermore, documentation in general is also a significant artifact. Documentation itself can take on many forms, as Ma et al. showed (Ma et al., 2018). If we assume a more broad definition of the term "documentation" things like productive developer discussions can also be part of this artifact class. Besides documentation, developer communication is an indispensable artifact that potentially provides important insight into the development process. This communication can take place via e-mail. But other channels, like instant messaging, are also important to investigate. Another important artifact mentioned in the cited work is version control data. Artifacts like Git-commits and logs contain important data on what and when changes were made to the software. In addition to that, already-mentioned artifacts like bug reports also constitute a category of software artifacts. Together with issues (or tickets), they can be summarized as issue data. Lastly, another category of software artifacts is build data. Build data means the log output and other artifacts created by build systems (servers) like "Jenkins" or "GitHub Actions". Exemplary artifacts can be defined as build steps or build logs.

All the listed artifacts can be part of the final classification system for insight into the inner source development process. They are summarized in Table 2.1.

Artifact Category	Exemplary Artifacts
Source Code	Code files, Test code files
Documentation	Design Documentation, Developer Discussions
Developer Communication	E-Mails, Instant messages
Version Control Data	Commits, Commit-Logs
Issue Data	Issues/Tickets, Bug Reports
Build Data	Build Steps, Build Logs

Table 2.1: Software Development Artifacts

2.2.2 Artifact Sources

With the artifacts identified, the problem of where the artifacts come from remains. In general, in software development, the listed artifacts come from a potentially large number of different sources. A major problem for the classification system therefore will be accumulating data from many distinct data sources.

For this type of data collection, different tools and libraries have been built. One of these tools is *GrimoireLab* (Dueñas et al., 2021). Tools like this can take a wide array of data sources and provide a pipeline for retrieval, analytics, and final reporting. In the cited GrimoireLab paper, different examples are given of the kinds of data sources that are possible to analyze. Table 2.2 shows the mentioned artifact categories and possible real-world sources for these artifacts. This shows how potentially broad the field of data sources can be when designing a comprehensive software development artifact classification system.

Artifact Class	Exemplary Data Sources
Source Code	GitHub, GitLab
Documentation	Confluence, GitHub README
Developer Communication	Slack, E-Mail-Client
Version Control Data	Git
Issue Data	GitHub, GitLab
Build Data	Jenkins, GitHub Actions

Table 2.2: Artifact Sources

2.2.3 Artifact Classes

Another central problem for the final design and implementation of the classification system will be the final classes of the software artifacts. As already mentioned, in the related work section (see section 2.1.1), various attempts at classifying software artifacts already exist. Often, these classifications are for a

narrow use case. Alqahtani and Rilling, for example, use tagging for security purposes and classify artifacts into classes like "Authentication Abuse" and "SQL Injection" (Alqahtani and Rilling, 2017). Another more focused approach was taken by Rane et al. who classified only source code comments and gave them classes like "summary" and "rationale" (Rani et al., 2021).

This thesis' take on software artifact classification is more general, considering different artifacts for the more generalized use case of software development analytics. Therefore, the classes for the artifacts also do not cater to a specialized use case. As a result, a core problem of the system will be properly classifying the artifacts into meaningful classes that are sufficient for the later inner source analysis.

For the inner source analysis, classifications are needed that make the development process more transparent. With this information, examplary questions about the development process can be formulated:

- What kind of artifacts are added to the software project?
- When are artifacts added to the software project?
- Are the added artifacts new or are they improvements to existing ones?
- Who added an artifact?

From these questions, different types of classifications are possible. What constitutes the added source code? Is it a comment or productive code? What do the developers messages and e-mails contain? Is it design documentation, source code, or, for example, a markdown artifact? Furthermore, by analyzing the version control logs, we can ask who committed an artifact, when did he or she do it, or was it an automated commit by a bot. All these questions represent possible classification possibilities that can be pursued. The goal will be to find a representative set of suitable classes for the stated artifacts to allow for deeper analytics of the underlying software development project.

When talking about the classification of software artifacts, it is important to differentiate between two important types of classification. All the mentioned artifacts consist of some kind of text. This text is either written in a formal language like a programming language or in an informal language like natural speech. Therefore, classification based on the artifact's text is possible. Furthermore, it would also be possible to find classes for the artifact not based on the textual content but from the metadata and context provided by the artifact. If we look at communication data, it is, for example, possible to differentiate between asynchronous and synchronous messages. Another example would be the incorporation of data like time to classify the artifact. Other processes on all kinds of metadata is possible for the creation of classifiers.

To keep this thesis focused, the design of the classification system will be limited to a text-based approach. Artifacts will still be categorized into the mentioned artifact categories (see table 2.1) but further classification will only be done via the textual content of the artifact. To illustrate the choice of classification, figure 2.1 shows the different kinds of classifications.

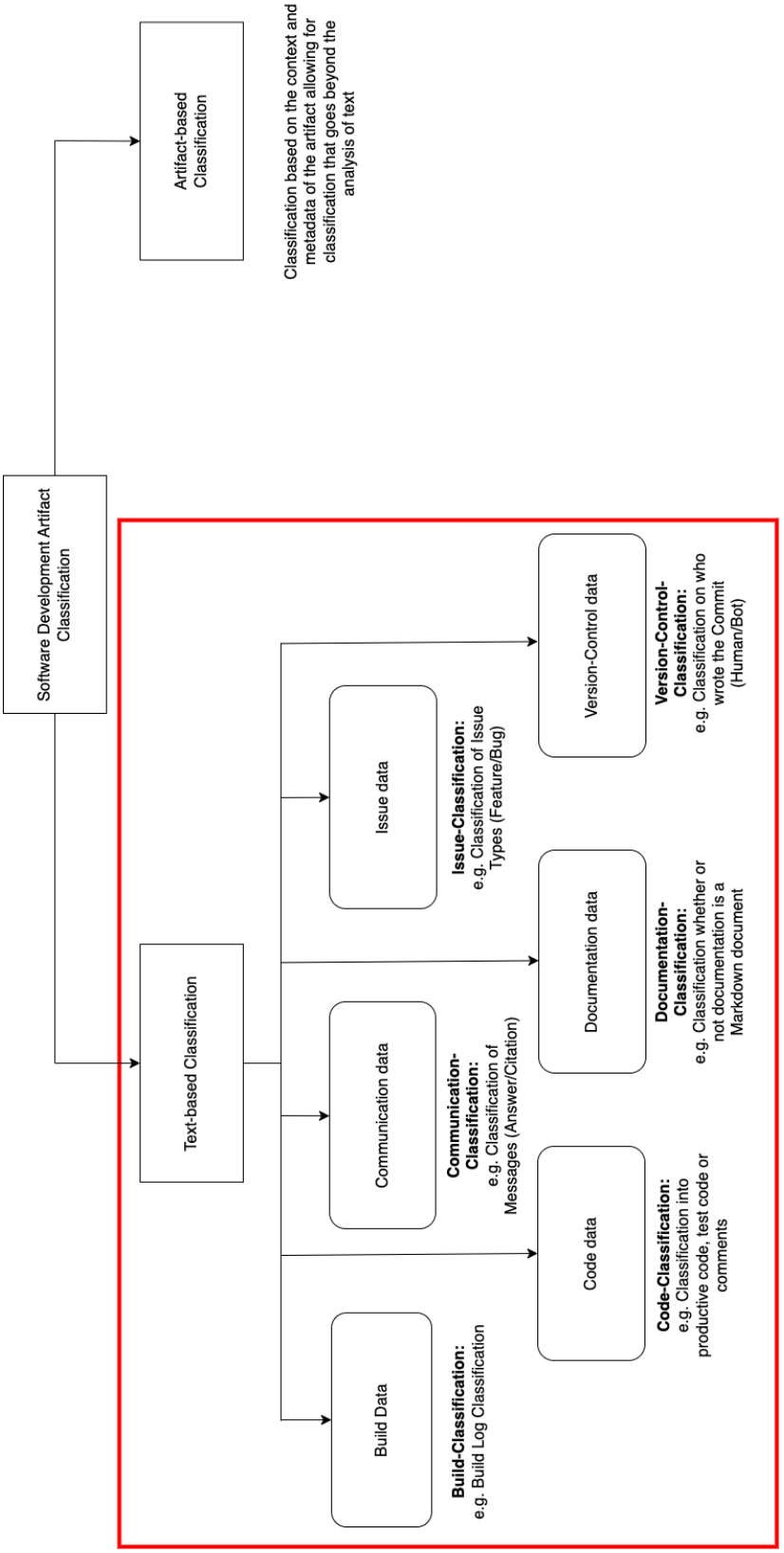

Figure 2.1: Artifact classification approaches

3 Objective definition

To put everything up until now together, the objectives of the classification system will be formulated. The different objectives will describe the different aspects of the system and will be used in a later evaluation. This evaluation will check whether the actual developed design meets the needs formulated in the problem identification.

At first, the classification system has to be able to classify various kinds of software development artifacts, as defined in section 2.2.1 of this thesis. This forms the basis of the whole software. Different artifacts, classified in a diverse manner, create the foundation that is needed for later analysis.

Furthermore, these artifacts have to be sourced from somewhere. As described in section 2.2.2, software artifacts come from a wide array of different sources. For that reason, a major objective of the classification system is the support for multiple data sources and the easy adaptability of new ones in order to be flexible.

Besides only being able to source different kinds of artifacts from different kinds of sources, the results of the classification system also have to meet a certain criterion. Section 2.2.3 showed the problem of meaningful resulting classes that are needed for the subsequent analysis. These useful resulting classes are another indispensable objective of the final software.

Lastly, as section 2.1.3 showed, it is important to make a clear distinction from machine learning when designing a viable solution for the classification system. In the interest of repeatability, these techniques cannot be used because the inner source analysis has to be precise for legal and accounting reasons.

To summarize the mentioned objectives, they are again listed in a shortened form:

1. The classification system is able to classify different kinds of software development artifacts.
2. The classification system is able to read data from various data sources and is not bound to a fixed set of sources.

3. The resulting classification results are usable within a wide range of assessment methods and analytics to support the inner source methodology.
4. The classification system does not use machine learning techniques for artifact classification in order to assure repeatability.

4 Solution Design and Implementation

After defining the problem and the objectives of the classification system, the design science methodology stipulates the design of the solution. Moreover, implementation details will be provided to show the feasibility of the solution design. To create a viable solution, the design process will be outlined first.

4.1 Design Process

For the creation of the classification system, the first step will be to specify the granularity of classifications that are possible. Certain classifications, for example, may not be entirely source-independent. It is therefore important to deal with this subject and decide what types of classifications are necessary for the final solution. Based on that knowledge, it will be possible to define the prerequisites and desired classification for the final system. This includes a comprehensive listing of artifacts and their respective possible classifications that are required by the classification system.

After defining this initial situation, it will be necessary to design an abstract data pipeline model that can solve the classification problem. It will define how the initial data will be processed and what steps will be taken to accomplish the set objectives.

The overview pipeline model will then be worked out with tangible processes and concrete tools that perform the actual classification of the software artifacts.

Finally, all the conceptual design models and processes will be implemented. In this part, implementation details about data retrieval, data processing, and data saving will be discussed.

Figure 4.1 shows the process.

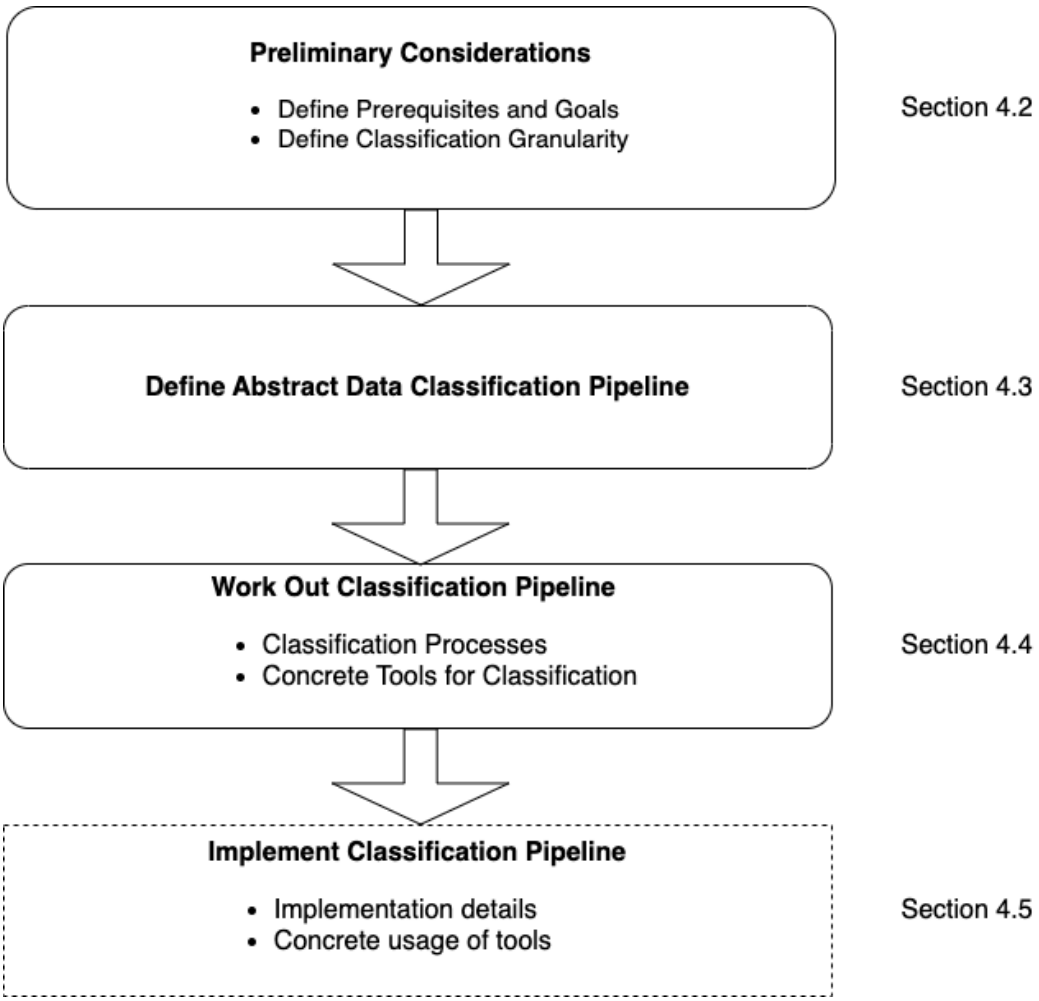

Figure 4.1: Design Process

4.2 Preliminary Considerations

The preliminary considerations will explain the issue of classification granularity and outline what prerequisites and goals are present for the classification system.

4.2.1 Classification Granularity

An important point to consider when defining the starting point of the classification system is the granularity of the classification. Granularity in the context of this thesis means how specific or specialized a classification is for a given artifact.

As part of the objective definition, it was mentioned that the overarching goal of the classification is to be able to handle many kinds of data sources. Section 2.2.2 showed the many possible data sources that can occur as part of the classification system. Because different data source systems may handle their textual data differently, the finest granularity for classification would be the "source-specific classification". To make this point clear, an example is given.

Developers in a team will use an instant messaging service. This service may be "Microsoft Teams", "Slack" or some other specific message service. The textual data created by these services can include source-specific codes or other keywords that may be needed for some classifications. An example would be the integration of username links within the text to tag a colleague. This tagging may be needed to classify communication artifacts into "is tagging person" and "is not tagging person". Through this example, it is also possible to see that a certain source-specific classification is only applicable to an artifact of a single artifact category. An instant messenger, for example, will only produce communication data.

The "source-specific classification" is the most granular type of classification. The "artifact-specific" or "source-independent classification" is one level above it. Just as the "source-specific classification" the "artifact-specific classification" will classify each artifact based on its artifact category that was defined in section 2.2.1 and figure 2.1 of this thesis. But instead of being tailored to a specific source, these classifications can be applied source-independently. A good example of this kind of classification would be the classification of source code into productive code, test code, and comments. Neither the programming language nor the repository holding the code matter when applying this classification if the right tools are used.

Apart from classifications having some kind of artifact category as their basis, the "artifact-overraching" or "general" classification form the least granular type of classification in the final system. Some classifications will be possible without knowing the category of the artifact or its source. This is, for example, the case for classifications like the inclusion of media or the usage of a formal or informal

language.

Figure 4.2 shows the different levels of granularity present in the classification system that will be designed. In addition, the graphic will include all the stated artifact categories.

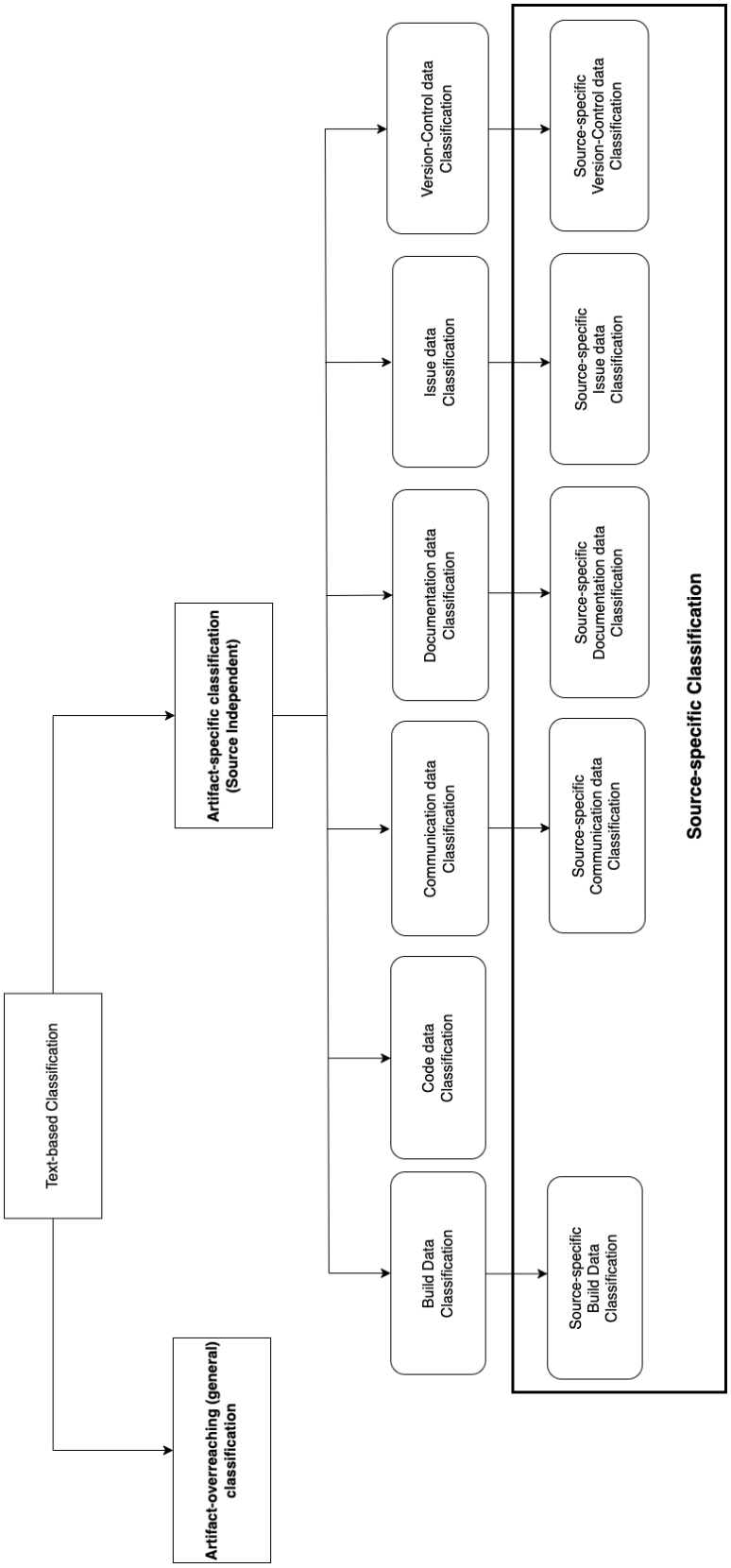

Figure 4.2: Classification Granularity

4.2.2 Prerequisites and Goals

Before designing the classification system, the actual prerequisites and goals of the system have to be specified. While many prerequisites and the overarching goal are already defined in chapters 2 and 3, a more detailed and binding specification for the system is needed to create a sufficient design and implementation.

Relevant Artifacts and their Sources

For the text-based classifications, a wide array of artifacts is used and classified. In chapter 2, different artifact categories and exemplary artifacts for each of them were given. This list already contains all the artifacts that will be interesting for the classification system. When implementing and designing the real solution for the classification, it will be relevant where the artifacts are from. In section 4.2.1 classification granularity is explained. Especially for the source-specific classification, concrete sources are important. Like the artifacts, the sources are also already defined in Section 2.2.2. Objective two in the objective definition says that the final classification system has to be able to process different kinds of data sources equally. That means that possibly not formerly known sources are also assimilable into the software system. Section 4.3 will build on this and explain how the data pipeline of the classification system deals with this issue.

Wanted Classifications

By knowing the artifacts and their sources, the input into the classification system is defined. The important next step is to understand the desired output of the classification system. Other than the sources and artifacts, chapter 2 only outlined possible classifications. Therefore, it is important to know the wanted classifications at each of the three granularity levels. The enumeration of these classifications will not be comprehensive. This thesis aims at presenting and designing an overview of different possible classifications to benefit inner source development. Creating a complete and comprehensive classification system that handles most possible classifications will not be part of this thesis. Therefore, the proposed classifications only represent a reduced subset of all possible classifications. The selection of the classifications is based on the related work and tools (section 2.1), and self-creation. The goal is to offer classifications for all artifact categories.

For the artifact-overreaching classification, only classifications can be applied that are applicable to every type of text. This includes what kind of text is used and if the text has some kind of inclusion that is generally detectable. For that, three different classifiers are proposed:

1. Usage of Markdown or other general markup language: Yes/No

2. Inclusion of media (pictures, videos, charts): Yes/No
3. Kind of Language: Formal/Informal
4. Inclusion of Special Characters: Yes/No

When looking at the artifact-specific classifications, the classifications will be grouped by their artifact category (see Figure 2.1). For each of the categories, multiple classifications are proposed. Like section 2.2.3 proposed, all classifications have to be text-based. Table 4.1 shows all proposed artifact-specific classifications.

The last granularity concerns source-specific classifications. These are also categorized into artifact categories, like the source-independent classifications. Because they are source-specific, most of these classifications use source-specific coding. Examples for this would be user tagging or other special syntax used to indicate other users or URLs. Within the team collaboration tool "Slack" (https://slack.com/), for example, liked users in a message are saved like this: "*<@U0214SVB5C1>*" where "U0214SVB5C1" is the user handle of a different person. Normal URL links are represented similarly (Example: "*<https://medium.com/@smotaal/when-i-ramble-180aba2256ee|medium>*"). Besides communication data via Slack, many kinds of text-specific coding can occur in different kinds of artifacts. What kind of artifacts are interesting and how to extract them for classification depends on the software project and the concrete tools used within a software development project.

MECE principle

An important principle for each classification in the classification system is the MECE principle. It stands for "mutually exclusive, collectively exhaustive" and is usually used in the context of consulting. In general, it is a principle that ensures that a set of items has to fulfill these two properties. The principle tries to achieve maximum clarity and completeness. (Rasiel, 1999)

Clarity and completeness are also important for the classification system, which implies these two consequences for each classifier that gets designed:

- Every classifier must not assign multiple classes to one artifact (mutually exclusive).
- Every classifier has to assign some class to every artifact (collectively exhaustive).

Artifact Category	Classifications
Code data	• **Type of Code**: Productive/-Comment • **Programming Language used**
Communication data	• **Type of Message**: General Conversation/Citation (Answer)
Documentation data	• **Type of Documentation Format**: General Text/Code/-Diagram/Table/Media
Issue data	• **Type of Issue**: Feature Request/Bug Report/Incident • **Inclusion of Stacktraces**: Yes/No
Version-Control data	• **Type of Issuer**: Bot-Commit/User-Commit • **Type of Commit**: Corrective/Adaptive/Perfective
Build data	• **Used Build Steps/Tools**: e.g. maven build • **Inclusion of Warnings/Errors/Information**

Table 4.1: Artifact-specific Classifications

4.3 Abstract Data Pipeline

To create the needed classification system, an abstract data pipeline is needed. This design is the foundation for further concrete implementation of the classification processes. For the design, a modified Extract-Transform-Load-Pipeline (ETL-Pipeline) is used. The individual steps consist of the following processes.

The extract step's responsibility is to receive all the relevant data that was defined. Different real artifact sources were already discussed in section 2.2.2 of this thesis. The challenge now is to find a viable process or tool for data extraction. Section 2.2.2 mentioned the application GrimoireLab, which contains different tools for development data retrieval. The tools that are used are called "Graal", "Perceval" and "Arthur" (Dueñas et al., 2021). In the later implementation of the classification system, an already-aggregated dataset from GrimoireLab will be used. This does not mean that the data retrieval cannot be done via a different process or tool. The final goal of this step is to aggregate the diverse data into a single usable output for further processing and final classification. The exact format of the data is not important as long as the following steps of the pipeline can process it.

The second step of the pipeline transforms the extracted data and performs the actual classification. For that, multiple sub-steps are necessary. At the beginning of the transformation step, data pre-processing may be necessary. This includes tokenization, parsing, or data cleaning of the textual development artifacts. Subsequently, the actual classification takes place. This classification will be done in a three-step process. Section 4.2.1 showed different classification granularities between different types of classifications. The three levels "artifact-overarching classification", "artifact-specific, source-independent classification" and "source-specific classification" represent the different classification steps in the pipeline. At first, the artifact-overarching classification takes place. The process for this will be the same for every artifact. Thereafter, artifact- but not source-specific classification can take place. At the very end, source-specific classification will be executed if necessary. After the classification is done, post-processing steps may be necessary.

The last step of the pipeline will be the loading of the final data product into some data sink via a data writer. What kind of data sink is used or how the final data product looks like is subject to the concrete implementation.

To give a better overview of the defined data pipeline, figure 4.3 shows the pipeline and its steps.

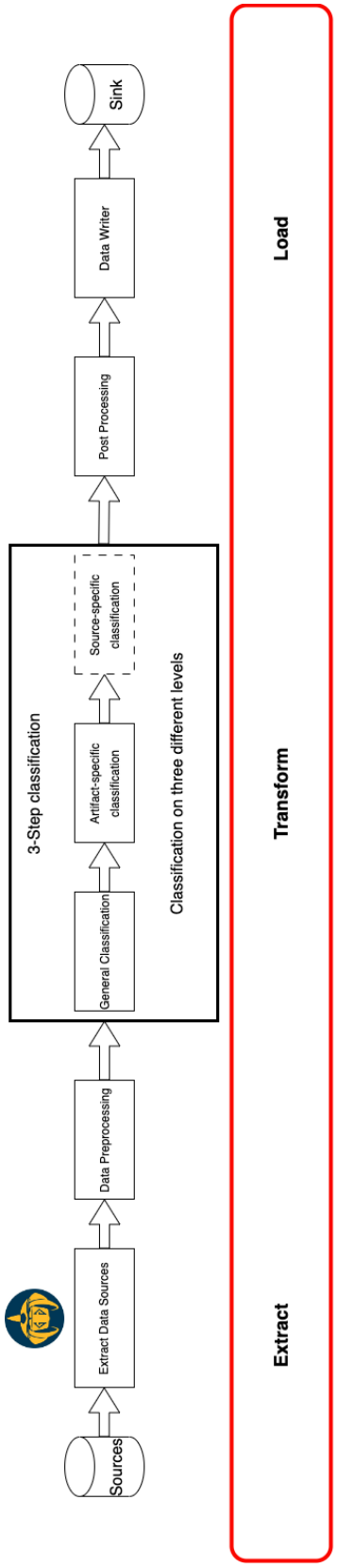

Figure 4.3: Abstract Data Pipeline

4.4 Processes and Tools for Artifact Classification

The abstract data pipeline outlined the processes needed for the classification system. Together with the knowledge about the artifacts, their sources, and the wanted classifications, real implementable processes need to be created.

4.4.1 Artifact Extraction and Pre-processing

For the first step, the data present in the source systems has to be extracted. As mentioned in section 4.3 this can be done by different means. Overall, it is important to aggregate the needed data into a usable format. Tools like "GrimoireLab" (Dueñas et al., 2021) can help with this task and provide a unified interface for the retrieval of different development artifact sources. The final format of the aggregated data will not be relevant for the rest of the classification, as long as the following steps of the pipeline can process it. This can, for example, be achieved via a unified and stable data API. Because of the potential polymorphism of this problem and its non-involvement with the actual relevant classification, further details will not be discussed. As already mentioned, the aggregated data for the implementation part of the thesis will come in the form of a relational database.

After aggregation, the data may have to be pre-processed. This pre-processing can include many steps like data cleaning, data removal, data expansion, further aggregation, or other processing steps. In the case of the classification system, the pre-processing includes two important steps. At first, much of the aggregated data gets discarded. In the aggregation process, a lot of different metadata gets collected. This data is not needed for further classification. Rather, for each table, a set of important metadata and the text to be analyzed have to be defined. As a result, a data table is created only containing a dataset that is meaningful for further processing. As a second pre-processing step, each data entry for a software artifact gets extended. Each entry gets three additional new fields or columns. The columns represent the collection of different classification labels based on the classification granularity mentioned in section 4.2.1. Therefore, three new columns will be added: "general classification", "source independent classification" and "source dependent classification". Figure 4.4 shows the pre-processing transformation of the original data into a relational model. It is important to understand that a relational model is only a possible example of the format of the data. Document stores with the JSON-format or other data representations are possible as long as the following classification steps in the pipeline can process the format.

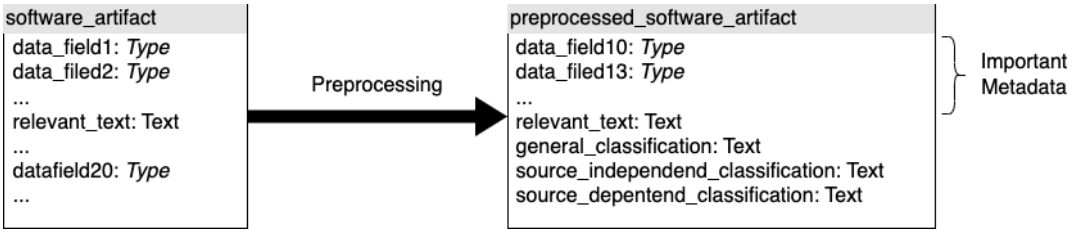

Figure 4.4: Data Pre-processing

4.4.2 Artifact Classification

The abstract pipeline defines three classification steps. Each of them holds different classifiers that can be part of the final pipeline for a certain artifact. Each of the classifiers will be described and grouped if a grouping is applicable. It is important to note that the concepts depicted here for each classifier are only one possible solution. For each classifier, different designs and concepts may be possible.

General Classification

General classification classifiers can be used with every textual software artifact and are therefore universally applicable. Four different general classifiers were proposed.

1. *Usage of Markdown*:
   For the classification of the usage of the Markdown language, a Markdown parser will be used. In section 2.1.4 the "Marked" parser gets mentioned, although other similar parsers like "Markdown-it" (https://github.com/markdown-it/markdown-it) are also usable. For classification, only the first step of the parser is needed. With the lexer, the single tokens of the text string can be extracted. Based on the used lexer, the tokens have to be analyzed. Because every simple text (text without any Markdown-specific keywords or key symbols) is automatically a valid Markdown document, an assumption has to be made for the classifier. If no other tokens than standard paragraphs are part of the text string, the text will not be classified as "Markdown". In all other cases, it will get this label. To accomplish that, an iteration over the tokens is necessary. If the name of a token does not match the standard paragraph, the text will get the "Markdown" label. It is important to mention that the names of the tokens and the utilization of the lexer are highly dependent on the parser used.
2. *Inclusion of Media*:
   To classify if a text artifact includes media inclusions, it is necessary to know when this classification is possible at all. Source code and other

formal language artifacts, in general, do not include media like photos or videos. This reduces the search to artifacts with natural language like documentation, issues, and communication. Different ways exist to include media in artifacts like these. As a simplification, the presented classifier again uses a Markdown parser and therefore assumes that documents of these artifact classes may be written in this format. Otherwise, another detector has to be used. Similar to the "Usage of Markdown" the lexer of the Markdown parser will be sufficient. For the inclusion of video and photos, special tokens exist that can be detected. The classification logic works similarly to the "Usage of Markdown classifier". But in this case, the relevant tokens are the media inclusion tokens.

3. *Inclusion of Special Characters*:
   To find special characters inside a text string, regular expressions can be used. At first, a correct regular expression for the classification is needed. The expression has to be able to match the defined special characters. The regular expression "*[\U0001F300-\U0001F6FF]|[\U0001F1E0-\U0001F1FF]*", for example, matches all possible emojis in a text. A package like the Python "re" package can be used to compile and match against the pattern. Unicode identifiers are used to identify the special characters. Based on that, many types of special character classifiers can be built. With the regular expression defined, it is possible to match a given text against it. If the matcher returns positive feedback, the text will get the corresponding special character label. Otherwise, it will get the label stating the opposite.

4. *Usage of a Formal Language* Classifying whether the present text artifact is a formal language poses the problem that formal languages are not only programming languages but also other kinds of languages like a markup language. Under the assumption that only a fixed set of formal languages is used within the software development artifacts, the parsers of the corresponding languages can be used. Each text artifact gets checked by each of the parsers. If a parser recognizes the text as part of its language, the corresponding label will be set. If none of the parsers match, it is assumed that the text is in natural language. For this process working parsers within the programming environment have to be available. The process of checking whether a text artifact matches one of the languages is then dependent on the output of each parser.

Artifact-specific Classification

Artifact-specific classifiers are only usable within their respective artifact category. With them various additional classifications are possible. The proposed classifiers will be described and grouped by their artifact category.

Code Data

- *Used Programming Language:*
  Detecting the used programming language without using machine learning is similar to the general "Usage of Formal Language" classifier. Instead of allowing any kind of formal language parser, only programming language compilers are allowed. The rest of the process works similarly. This classifier is therefore a specialization of the "Formal Language" classifier. The only difference within it is that it knows more than two classes and only uses parsers for real programming languages. Parsers for markup languages or similar formal languages are not allowed. If one of the parsers detects the text string as part of its language, the corresponding label of the language will be set. If none of the parsers succeed, the code artifacts get classified with the "not known programming language" label.

- *Type of Code:*
  For detecting the type of code (Code/Comment) the already mentioned tool "cloc" can be used. With this command-line tool, it is possible to count the lines of comment and real code in a source code file. Given a text artifact and the corresponding programming language, it can be determined if the text includes code, comments, or both. Knowing the programming language of the text artifact is important because otherwise "cloc" will not be able to process the source code. Getting that information can be done using the same process as the "Used Programming Language" classifier. This means that a fixed set of parsers first determines what kind of language is present. If no programming language can be detected, either a fallback language is defined or the classification defaults to "code". Because of the artifact category it is known that the text has to be some kind of code at least. For the classification itself, the text artifact will be converted into a temporary file that is then analyzed. "cloc" offers multiple output options that are machine-readable. For example, JSON can be used. The JSON properties "comment" and "code" then show how many lines each original artifact contains. For the final labeling, different strategies can be used. To fulfill the MECE criteria the artifact gets the "code" label when the majority of lines are code. The label "comment" is set if the majority of lines are comments. If both counts are the same, the label "code" is set. Other strategies for labeling are also possible and depend on the preferences of the underlying project. For example, labeling the artifact a "comment" could only be done if more than two-thirds of the lines are comments.

Communication Data

- *Type of Message:*
  To classify the type of message (Normal Message/Answer/Forward) it is

first necessary to specify the kind of message being analyzed. The communication data is defined in section 2.2.1 as either e-mail or direct instant messages. Because instant messages heavily rely on source-dependent tokens and text structures for forwarding and replying, only e-mails will be classified. For that, the e-mail subject is used to allow for a text-based classifier. Most e-mail clients automatically add prefixes to forward and answer e-mails. Examples of these prefixes are "Fwd:" for forwarding and "Re:" for answering. For the classifier, multiple variants for these tokens can be defined. With these tokens defined, it is possible to check whether the subject text starts with one of those tokens or not. If it does, the corresponding label can be set. Otherwise, the e-mail is classified as a "normal" e-mail or message to always provide a suitable class.

Documentation Data

- *Type of Documentation Form:*
  Documentation data can take many forms and come from many various sources. It can be stored in proprietary wiki software like "Confluence" (https://www.atlassian.com/de/software/confluence) or in single loose documents like Markdown or text documents. While classifying the content of proprietary wiki software constitutes source-specific classification, Markdown documents offer easy insight into what kind of text and media are used within a documentation document. General classification already showed how to detect media like photos and videos. Additionally, code, diagrams, and tables can also be detected because they are usable in Markdown as tokens. With the same parser and lexer process as before, the classification of single artifacts is possible.

Issue Data

- *Type of Issue:*
  The "Type of Issue" includes the three different classes "Feature Request", "Bug Report" and "Incident". The workflow of the classification is similar to the "Type of Message" classification. Several keywords are defined for each class of issue. How these keywords are defined depends on the software project that gets analyzed and its convention regarding issue texts. To generalize this, for this thesis, examples for each class will be given. Books like Helmut Balzert's "Lehrbuch der Softwaretechnik" show how natural language requirements can be formulated. (Balzert, 2009) Some keywords used in this book were translated and used in this design.
  - Feature Request: "the system has to", "as a <role>", "criteria", "feature"

  - Bug Report: "expected behavior", "observed behavior", "bug"
  - Incident: "failure", "shutdown", "incident"

  Many more keywords are possible, and the examples do not represent a comprehensive list. With the keywords defined, a keyword search can be conducted on the issue text and title if they are available. The keyword search itself only checks whether the given text includes one of the given keywords. If a class matches, the text artifact gets the corresponding class or label. If none of the keywords match, the issue gets labeled "unspecific". With this, a clear classification is assured.

- *Inclusion of Stack Traces*
  Stack traces and their appearance heavily depend on the underlying programming language. Because of this, a general classification of stack traces for all possible stack traces is not possible. Similarly to the classification of programming languages, each language has to be dealt with separately. By creating a regular expression representing a stack trace for each needed programming language, each issue text can be matched against these. This procedure is the same as with the special characters. If a stack trace regular expression matches the given issue text, the issue can be classified as having a stack trace within it. Because the stack traces are unique to each language, the language can also be identified and added to the classification. If none of the regular expressions apply to the given issue, it is assumed that no stack traces are included in the text. This is only a wrong assumption if the stack trace present is not part of the set of regular expressions that are part of the classifier. Therefore, a configuration for each software project is needed to ensure that the classifier can detect all relevant stack traces.

Version-Control Data

- *Type of Issuer:*
  Commits in version-control systems can have different types of issuers. They are either directly created by a human or automatically made by a bot. An example for bot-made commits is the automation of certain tasks with a build and automation server software like "Jenkins" (www.jenkins.io). Because these commits are automated, they follow a given format. As an example, it will be assumed that a bot commit provides a certain prefix in the commit message. This prefix can take any possible form. An example would be "*[Bot]:*". Again, the final configuration of a classifier depends on the project and the chosen conventions. With the prefix at hand, it is possible to classify the commit via its message. This works the same way as the "Type of Message" classifier in the communication data category. To ensure classification, every commit that is not identified as a bot commit is

automatically a human commit as a fall back. The label "Bot commit" as a fallback would also be possible.

- *Type of Commit:*
  The type of commit describes what kind of changes have been made to the underlying source code. To classify this, the commit message of the commit artifact is used. Similar to the "Type of Issue" classification, keywords will be used to identify the classes. Depending on the software project and the development team, these keywords may vary. The final classes for the type of commit are "Adaptive", "Perfective" and "Corrective". These classes were defined by Sarwar et al. and describe actions like creating new features (Adaptive), refactoring the application (Perfective) or fixing a bug (Corrective) (Sarwar et al., 2020). The classes are a subset of the presented commit classes in the related work section by Hindle et al. (Hindle et al., 2009) For each of the classes, the following example keywords are given:
  - Adaptive: "implement", "merge"
  - Corrective: "fix", "improve"
  - Perfective: "refactor", "clean"

  All example keywords are verbs conforming to a standardized way of writing Git commits. For example, Git itself writes its merge commit messages in the imperative mood with the verb "merge" at the beginning of the message. Other conventions for commit messages or similar artifacts may need other keywords. Sarwar et al. also present rules on when to add a commit to a certain class. (Sarwar et al., 2020)

  In the case of multiple matching classes for one commit message a hierarchy of the classes has to be established showing what class is preferable. Alternatively other preference systems can be implemented taking the keywords itself or the keyword count into account.

Build Data

- *Build Steps/Tools Used:*
  Section 2.2.1 mentioned build logs as a key textual artifact for software builds. They are created by a build server like "Jenkins". From these logs, it is possible to extract the build steps and tools used for them. With this information, it is possible to classify building artifacts based on which steps were part of their execution. Together with the steps, the tools used are also easily extractable.

  The log of the build server usually shows the executed commands and their output as a sequential text. By extracting the used commands, the single

steps of the build or deployment process can be detected. For example, the usage of "Maven" (https://maven.apache.org/) with the command "*mvn*" indicates the application build step of a Java (or JVM) application. Other commands like "*docker*" show the usage of containerization. With this information, a set of build and deployment tools and their CLI commands can be assembled. By scanning the log for these keywords, the build steps and their tools can be found, and the artifact can be labeled. To fulfill the MECE criteria the created classes have to be disjoint. This can be done, for example, by creating a classifier for every step and having the classes "included step x" or "does not included step x".

- *Inclusion of Warnings, Errors, and Information:*
  Another important part of the build log is not only the executed commands but the actual output of these executions. They contain information about build and deployment errors, warnings, and important info that is usable in later analysis. For analyzing the log about this a log parser can be used. Section 2.1.4 mentioned the Jenkins "Log Parser". It is a plugin that extracts errors, warnings, and info logs. When modifying the code to run it outside the Jenkins environment, the tool can be used for the desired classification. Depending on the output of the tool, a build artifact gets one of the mentioned labels. Which class is assigned when the artifact contains different kinds of logs, depends on the preferences of the software project and classification system. If the tool does not recognize any of the log types in the artifact, the label "No Information" will be applied.

Source-specific Classification

Source-specific classification is very individual and requires knowledge about the exact data source and how it performs certain text codings. In general, every kind of artifact category except code data can inhabit a source-specific classification. Code data cannot be source-specific because source code can only be dependable on the specification of the programming language. The examples of Slack user handles and Slack links as real source-dependency were already mentioned. In theory, every kind of classification is possible as long as the source system allows for the feature in its text artifacts. Confluence was already referred to as a documentation wiki system. As part of a possible classification, single wiki pages could be analyzed as artifacts and checked for so-called "Macros". "Macros" are components that form part of a wiki page text that can be used for inserting music and video or editable diagrams with a "draw.io" editor. Detecting certain relevant macros could be a possible classification for an artifact. (https://confluence.atlassian.com/doc/macros-139387.html)

These examples show the possibility of a diverse set of classifiers if the source systems are known. This is especially interesting because fewer assumptions

have to be made regarding the artifact. When detecting media inclusion, it was assumed that the given document conforms to Markdown syntax. Knowing the source, such assumptions do not have to be made. If, for example, documentation data artifacts get classified, the format of the text will be clear if the data source (e.g. Confluence) specifies the format.

In the later implementation, one source-specific classification will be implemented. The classification of whether another user is linked to a Slack instant message can be easily implemented. Similar to other already-designed classifiers, regular expressions will be used. For the user handle of slack, the regular expression can be represented with this string: "*<@U[A-Z0-9]+>*". The task of the classifier now is to test if the given text artifact includes a substring that matches the regular expression. If a substring is found, the class "Includes Slack User Handle" can be applied. Otherwise, the class "Does not include Slack User Handle" has to be set. Simple examples like these show how a source-specific classifier can look like.

4.4.3 Grouping of Classifiers

With the classifiers defined, further conclusions can be drawn. All the classifiers can be grouped into different classifier categories. A group describes a group of classifiers working similarly or using the same strategy to classify software artifacts. A group does not imply the classification of similar content or the classification into similar classes. Because these categories look at the inner workings of the single classifier components, they can be useful for the implementation hinting at programming concepts like inheritance.

Four different groups can be identified:

1. Parser-based classification: Classifier using a parser or the parser-internal lexer to analyze a given text. The text can be source code or other formally defined text, like a markup language.
2. Regular expression-based classification: The classifier uses a regular expression to match against the text artifact to find certain inclusions inside the text or test the whole text.
3. Tool-based classification: This classifier uses an external tool to analyze the given text artifact. With the analysis, a classification is possible.
4. Keyword-based classification: Given or configured keywords are used in this classifier to search for them inside the given text artifact. Specialized versions of this classifier can define that the keyword is only a prefix or a postfix. Depending on the result of the search, a classification is possible.

With the four groups given, it is possible to group the defined classifiers and

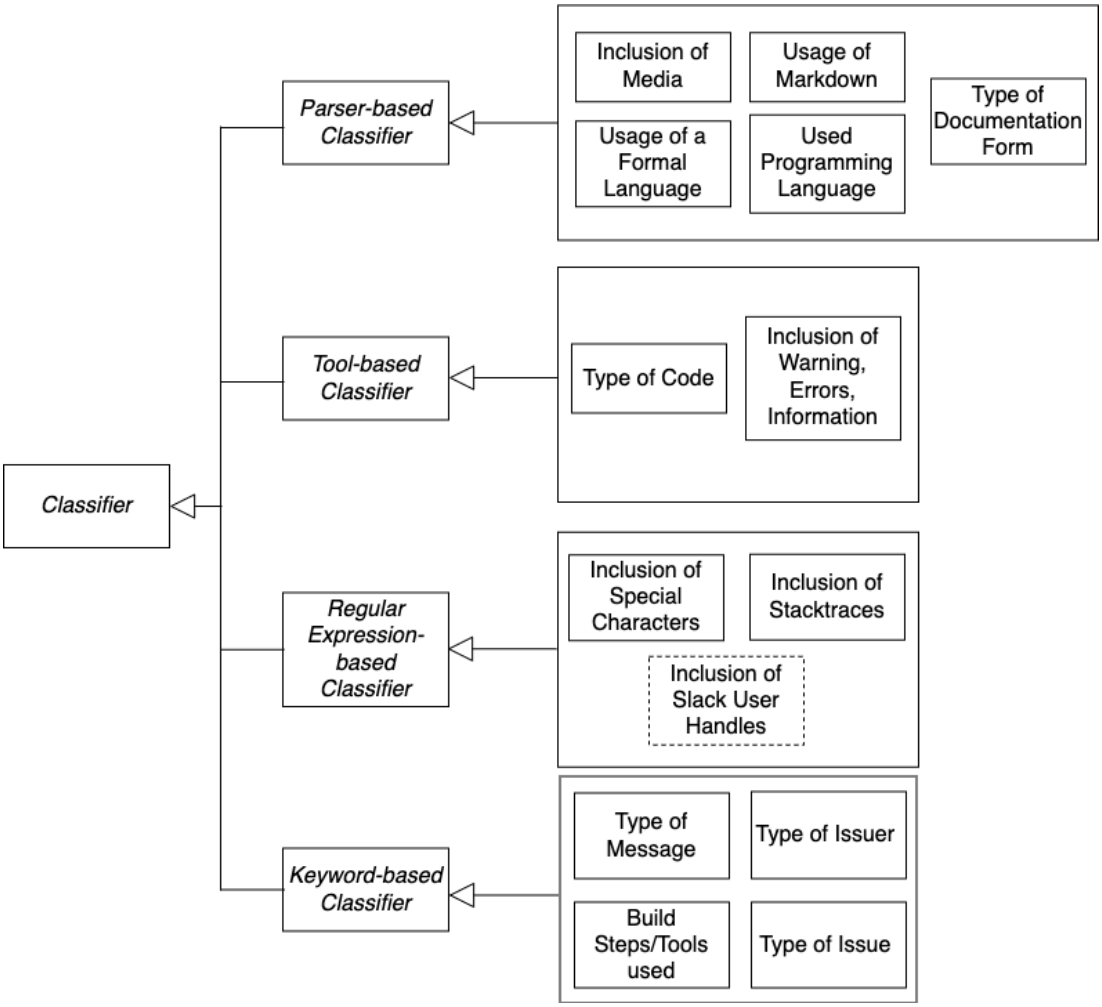

Figure 4.5: Grouping of defined classifiers

depict them in a diagram. Figure 4.4 shows the grouped classifiers in a tree diagram, signaling their common processes. Because this thesis does not offer a complete set of classifiers, other groups may exist that are not mentioned here. The classifier skirted with a dotted line is an example of a source-specific classifier.

4.4.4 Post-processing and Data Writing

The last part of the data pipeline consists of post-processing and writing the final data into a data sink. Both steps are specific to the software project the pipeline is used for.

Like the pre-processing step, post-processing can include many steps in editing the final classified data. The most important step is making sure that the final data can be written by the data writer into the sink. For that, depending on

the data sink and its type of persistence, data-changing operations have to be undertaken. For example, when using a relational schema, the data has to fit the sink table or even multiple tables. This problem will also be part of the example implementation. When the data sink has a different way of persisting data, other post-processing steps are necessary.

With the data in the right final format, the data writer finally persists the classified data. How this is done, again, depends on the concrete sink. The following implementation will give an example for this process, too.

4.4.5 Modular Pipelines for Different Artifacts

With the post-processing and data writing steps, the pipeline is complete. Now it is possible to create pipelines for different artifacts. This is necessary because artifacts from different artifact categories can only be used with certain classifiers. This implies that all classifiers have to be single, non-dependent components that can be used interchangeably.

For general classifications, this is not relevant, as they are intentionally usable on all incoming artifacts. As a consequence, they start the classification pipeline, as Figure 4.3 already showed. Artifact- and source-specific classification, on the other hand, poses the challenge of having to be modular for different artifacts. Thus, the final classification system has multiple pipelines serving different artifacts. In the end, every artifact category needs its own pipeline consisting of the general classification, its respective artifact-specific classifiers, and the applicable source-specific classifiers. As an option, post-processing and data writing can be handled differently for each of the pipelines.

To show how the artifact categories affect the pipeline, figure 4.6 shows two different pipelines. The upper pipeline shows the process for code data artifacts, and the lower pipeline depicts the process for communication data.

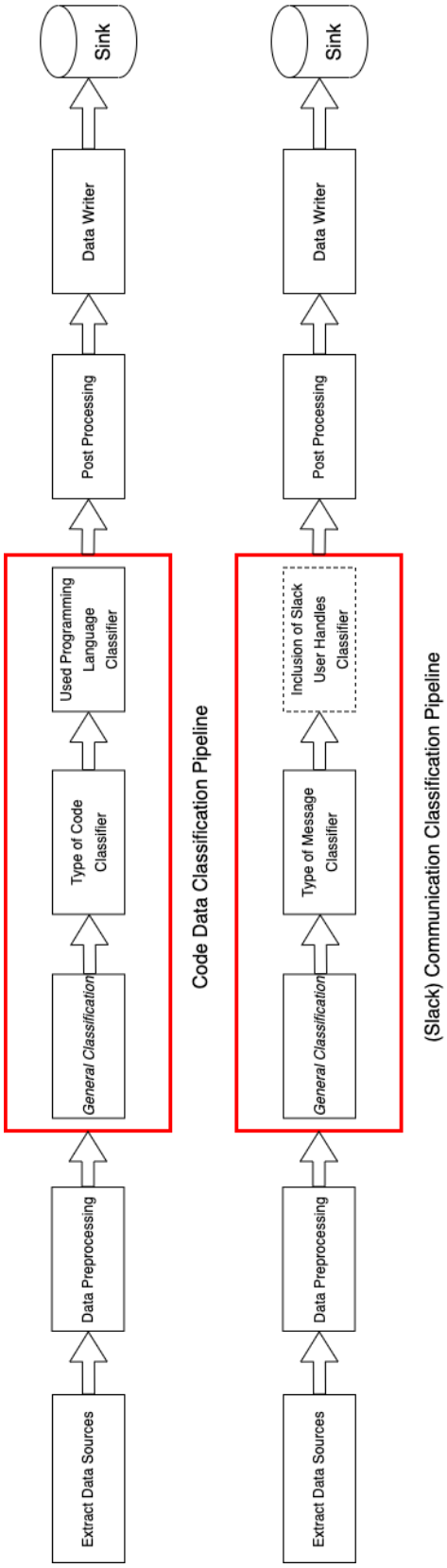

Figure 4.6: Exemplary Modular Pipelines for Code and Communication Data

4.5 Implementation Details

The designed system with its pipeline architecture and classifiers needs to be implemented for a demonstration to evaluate the design based on the objectives that were set. The presented implementation will include a representative subset of classifiers that were previously described. Each of the classifier groups will be part of the final implementation.

The implementation description will include the pipeline framework itself, the implementation concept for all classifier groups, pre- and post-processing steps, and data reading and writing. Initially, all used technologies will be presented. Implementation with other technologies in general is possible.

4.5.1 Used Technologies

For the implementation, some additional technologies are needed, and the programming environment has to be defined. Additional tools that are used for the classification are either already mentioned or will be presented in context with the pipeline component.

PostgreSQL

For data extraction and data writing, a relational database will be used within the implementation. This chapter already referred to the needed data for the demonstration. The used database will be a PostgreSQL (https://www.postgresql.org/) database running inside a Docker container (https://www.docker.com/). PostgreSQL is an open-source relational database management system that will host the example data and, finally, the classified artifacts. Because PostgreSQL is a widely used database system, integration into many programming languages is supported.

Python and Pandas

Besides the database, a general purpose programming language is needed for the creation of the classification pipeline. For that, the language Python is used with the interpreter version 3.9. Besides Python itself, an important part of the implementation is the usage of the Pandas library (https://pandas.pydata.org/). Pandas is an open-source data analysis and manipulation library that provides easy handling of the relational data that gets extracted from the PostgreSQL database. With Pandas Data Frames, manipulation and writing of the classification data can be handled. In addition to that, the PostgreSQL driver "psycopg2" and "sqlalchemy" for data writing are used to access the database.

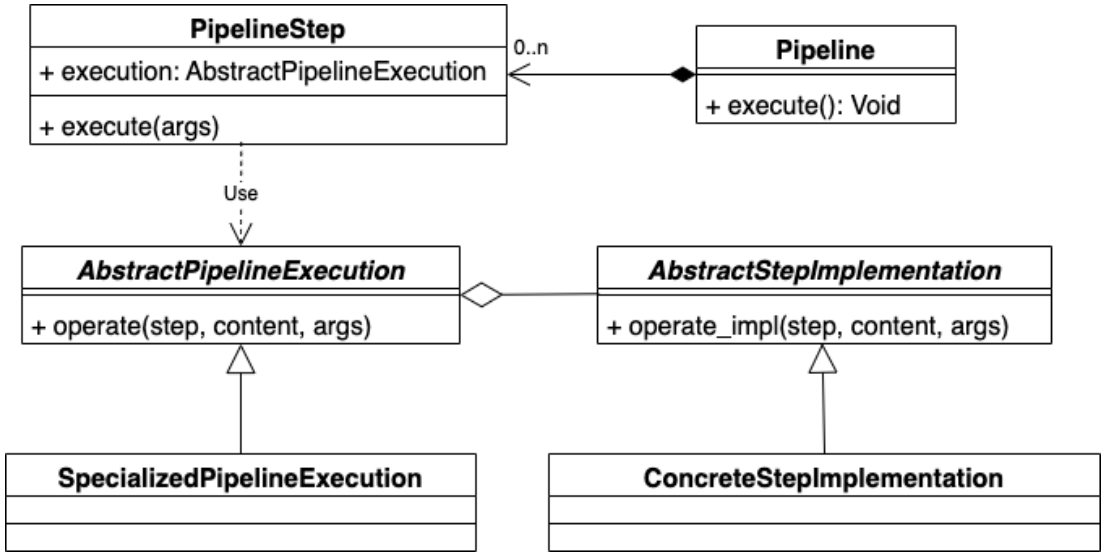

Figure 4.7: Class Diagram: Pipeline Structure

4.5.2 General Classification Pipeline

Before describing the implementation of the classifiers, the general functionality of the classification pipeline has to be stated. To assure the modularity of the pipeline and easy interchangeability of the single components, a generic pipeline is defined that holds pipeline steps. Each of the steps itself is a generic object that holds a concrete pipeline step and information on which method to execute on that concrete object when the pipeline triggers the pipeline step. Additionally, the pipeline step gets an array of optional arguments that will be transferred to the actual execution object.

The concrete pipeline step (pipeline step implementation) forms together with other classes in the solution a so-called Bridge pattern. The Bridge pattern is part of the "Gang of Four" design patterns (Gamma et al., 1994). It is used to separate the abstraction hierarchy from the implementation hierarchy. Within the pipeline, this helps to separate the pipeline steps from their concrete execution objects and their naming of methods, which eases the configuration and modularity of the pipeline that is needed for the different kinds of artifacts. To show how the class construct of the pipeline works, figure 4.7 shows a class diagram depicting the structure.

Here it can be seen that a pipeline is composited out of n pipeline steps. These pipeline steps hold a pipeline execution object that itself holds the concrete implementation of the pipeline step. The execution object and the implementation have their own hierarchies. When a pipeline step gets executed, it calls the specified method of the pipeline execution object (here: *operate()*). The execution object then triggers the actual implementation within its step implementation ob-

ject. The *operate*-methods themselves receive the pipeline step object of which they are part and the mentioned array of optional arguments. At the end of a pipeline step's execution, the next step gets triggered immediately. Otherwise, the pipeline comes to an end.

For the *ConcreteStepImplementation* itself this implies the following pseudocode for its execution:

```
class ConcreteStepImplementation(AbstractStepImplementation):
    def operate_impl(step, args):
        # Execute pipeline step with args
        step.next(modified_args)
```

Listing 4.1: Step Implementation Pseudocode

The following sections will go through each of the implemented pipeline steps and show how each concrete implementation works. It is important to note that each pipeline step within the implementation falls into one of three categories:

1. Data Requestor
2. Data Transformer
3. Data Writer

These categories represent specializations of the abstract pipeline execution class. The requestor produces data without other input besides its configuration, the transformer takes data and modifies it for further processing and the writer gets data for persisting it without passing it on to further steps.

4.5.3 Data Extraction and Pre-processing

Data extraction and pre-processing is the first step of every classification pipeline for every kind of artifact. They represent two fundamental step implementations.

The data extraction step falls into the data requestor category. Its purpose is to query the PostgreSQL database and provide the data as a Pandas data frame. Because different artifact data is stored in different database tables, the pipeline step is configurable via the list of arguments. For the configuration, a dictionary (map) has to be provided specifying the tables and wanted columns from the original table. Because column selection reduces the amount of processed data, the data requestor is also part of the pre-processing process. To achieve this task, a database connection via "psycopg2" gets established. The data is read with the Pandas "*pd.read_sql()*"-method. In the end, the data gets stored in a data frame and can be processed in the next step.

This next step is a data transformer, whose task it is to extend the data model. Missing from the data frame are the columns documenting the set labels for each

granularity of classification. Therefore, the three columns "general_classification", "source_independent_classification" and "source_specific_classification" are added to the frame and get populated with *None* in every data entry. This is done by a data transformer taking the data of the requestor. This transformer implements the abstract step implementation "TransformerImplementation".

4.5.4 Classifier Implementation

Each classifier that can be used in the classification pipeline is a transformer. Because of this, they are all specializations of the same abstract step implementation as the data model appender pre-processing step. Therefore, the method signature of every classifier follows the same pattern:

```
def transform_data(
    self, pipeline_step, data_to_transform, details
)
```

Listing 4.2: Classifier Method Signature

The *data_to_transform* parameter holds the received data from the previous step as a Pandas data frame. Every entry in this data frame holds a single software artifact that gets to be classified. As a consequence, every classifier iterates over the data frame, handling every entry separately. In the following implementation descriptions, only the classification of a single entry will be shown.

Parser-based Classification

From the group of parser-based classifiers, every classifier got implemented. Within the group, it is possible to distinguish between two kinds of parser-based classifiers. The "Inclusion of Media" and "Usage of Markdown" classifiers use the (Markdown) parser to detect special tokens within the text. Both are also general classifiers. The "Usage of a Formal Language" and "Used Programming Language" classifiers, on the other hand, use the parsers to validate their output to decide on the classification of the underlying artifact.

For the first kind, the process in the implementation works with the Markdown parser Markdown-It (https://github.com/executablebooks/markdown-it-py) that can be imported into Python. The *parse()*-method of the Markdown-It parser is then used to analyze the given text. This creates a list of tokens which have a "type" property. Depending on the classifier these tokens are matched to a given set of token types that indicate media inclusion or no non-standard text. If such tokens are found, the respective classification label gets set.

The two remaining classifiers are implemented by using different parsers and checking whether the parsing completes successfully. For the formal language classifier, this means using different parsers as Python packages and executing

the parsing. If a parsing error is raised, the text does not match the language. Therefore, if a parser does not raise an error, the text has to conform to the grammar of a specified formal language. The same principle also applies to the programming language classifier, while only programming language parsers are allowed. If a parser like the Markdown parser is used, the same logic cannot be used because every text counts as a Markdown text. As a consequence, the logic of the "Usage of Markdown" classifier is used to differentiate if Markdown is present or not.

Regular expression-based Classification

Regular expression-based classifiers work in a simple way. All the classifiers mentioned in section 4.4 were implemented. The general functionality is achieved by providing regular expressions as global variables. The handling of these expressions is done by the "*re*" Python package. To find out if a text part matches one of the given regular expressions, the methods *pattern.find_all(text)* or *re.search(reg_ex, text)* can be used. Depending on the method, a conditional statement can be formed, checking whether the regular expression is matched. Based on this, a classification can be made.

Tool-based Classification

As a proof of concept, only the "Type of Code" classifier was implemented from the tool-based classification group. For that, the "cloc" command-line tool was utilized. To make use of it within the program, the *subprocess* Python package was imported, which allows the execution of command-line tools within a Python program. As described, the classifier or the tool needs the underlying programming language to work properly. Either the programming language is known and can be received via the optional arguments or the language has to be detected. This is possible with the same process as the "Used Programming Language" classifier.

The usage of *cloc* then works as follows. The command to execute is formulated as a string. This command includes options about the used programming language (eg. *–force-lang=Python*) and the output format (*–json*). For execution, the text artifacts get wrapped in a temporary file to be used with the command-line tool. Subsequently, the JSON output gets parsed and can be queried like a Python dictionary. This is possible via the *json* Python package. With the dictionary, information for lines of code and lines of comment is available.

The classifier for the inclusion of warnings, errors, and information in build log files would have to be implemented in another way because the presented Jenkins log parser has to be modified to run within the Python pipeline.

Keyword-based Classification

In the last group of classifiers, the "Type of Message", "Type of Issuer" and "Type of Issue" classifiers were implemented. Because all of these data transformers behave in the same way, the presentation of all different classifiers is not necessary for the implementation design description.

The implementation of the keyword-based classifier is simple. For every class, an array of keywords gets defined in a global variable. While iterating over the single artifacts, every array gets checked to see if the text includes one of the keywords. This can be done in Python with the *in* operator. If a class of keywords matches, the class label gets set. Possible variations of the implementations may put the checking of the classes in a certain order. This is important because if a class is set, no other class can be set because of the MECE principle. Further alternative implementations may include a voting system where the class with the most matched keywords gets set.

The implementation with ordered class checks looks like this and can be applied to all mentioned keyword-based classifiers:

```
def transform_data(
    self, pipeline_step, data_to_transform, details
):
    for index, row in data_to_transform.iterrows():
        text = row[details].lower()
        if text is None:
            continue

        class_found = False

        for key in keyword_group1:
            if key in text:
                add_class("1")
                class_found = True
                break

        for key in keyword_group2:
            if key in text and not class_found:
                add_class("2")
                class_found = True
                break

        # Possible further classes

        if class_found is False:
            add_class("Fallback")

        pipeline_step.next([data_to_transform])
```

Listing 4.3: Keyword Classifier

4.5.5 Post-processing and Data Writing

The post-processing within the implementation consists only of one transformation step. This step is needed for the modification of the final classified data frame because it needs to fit the sink table in the database. For that, two operations are configurable via the optional arguments.

1. Deletion of table columns
2. Renaming of table columns

The deletion of columns can be useful if the final persisted and classified schema has to be reduced because certain queried metadata is no longer useful. The deletion itself is done via the *drop*-method provided by the Pandas data frame.

Because the column names coming from the source tables may have names that are not wanted in the sink table, renaming columns is possible. By providing a dictionary in the optional arguments that has columns to rename as keys and their new names as values, the *rename*-method of the Pandas data frame can be used.

Data writing is the last step in the classification pipeline and the only step in the data writer category. Because the pre-processing step already made sure the data frame could be written to the database, this step has no further responsibilities. "sqlalchemy" and the "psycopg2" driver are used to configure a connection to the database. Thereafter, the data frame gets written to the database with the *to_sql*-method.

4.5.6 Modular Pipeline Creation and Execution

With the creation of all parts of the pipeline, it is possible to assemble it. Because of the bridge pattern, this step can be done easily. At first, a pipeline object has to be created and then populated with pipeline steps. What pipeline steps are used depend on the artifact category and the exact source if source-specific classifiers are used.

For the easy addition of pipeline steps, a special utility method was created that takes a concrete step implementation, the pipeline object, the method name of the pipeline execution object, and the optional arguments for the step that configures it. With these resources at hand, the following method can be created:

```
def add_pipeline_step(
    input_pipeline, step_implementation, method, args
):
    implementation = None

    if isinstance(
        step_implementation, TransformerImplementation
        ):
        implementation = GeneralClassificationTransformer(
            step_implementation
            )
    elif isinstance(
        step_implementation, RequestImplementation
        ):
        implementation = SyncRequest(step_implementation)
    elif isinstance(
        step_implementation, WriterImplementation
        ):
        implementation = Writer(step_implementation)

    if implementation is None:
        print("step could not be added")
        return

    step = PipelineStep(implementation, method, args)

    input_pipeline.add(step)
```

Listing 4.4: Pipeline Step Addtion Method

5 Demonstration

With the implementation, it is possible to demonstrate the functionality of the design and its effectiveness. For that, a sample dataset will be used that gets classified by the presented implementation. Besides the dataset, the configuration of the different pipelines will be shown. At last, the results of the classification will be displayed.

5.1 Data Sources and Demonstration Dataset

Chapter 4 mentioned the usage of a relational database as the basis for the classification pipeline and system in general. This was done to abstract the data retrieval for this thesis. The relational database in question holds data from the GrimoireLab project. This data is publicly available and consists of different data tables representing software artifacts from the project. The used datasets are:

- **Git data**: Version-control data via the Git system. Each data entry represents a Git commit, stating its message, author, time, and other metadata.
- **Issue data**: Data from project issues and their comments. Every entry has the issue title, and every comment entry holds the text of the comment. Additional metadata, like usernames and creation dates, is also available.
- **Mail data**: Table holding data about individual e-mails sent within the project. The subject and a short extract of the mail body, together with metadata, are provided.
- **Slack data**: Data of individual Slack messages. Together with the message, metadata, like the chat channel and author, is also retrievable.

The GrimoireLab dataset does not include code data. Therefore, the sample code data will be provided by other sources. Similar to the other datasets, the code data will be stored in an extra database table. Because this demonstration does not require additional metadata for classification, none will be provided. For the code data, three different programming languages are used: Java, C, and Python. For each of the languages, different example sources were used. The "Java Design

Patterns" website (https://java-design-patterns.com/) offers examples for many design patterns and other snippets written in Java. Some of these patterns and snippets were used as text artifacts for the classification system. The same was also done for the C examples. For them, the website "w3resources" was used (https://www.w3resource.com). For the Python samples, snippets of the code written for the classification system were used.

5.2 Configuration of Classification Pipelines

For each of the datasets, twenty entries at random will be retrieved to demonstrate the functionality. This will be achieved by limiting the Postgres requestor to twenty entries. It is, of course, possible to classify all entries in the database in each dataset. This would produce several thousand classifications that are not practical to present as part of this thesis. The only dataset not consisting of several thousand entries is the code data table. For demonstration, only twenty entries in this table are available.

Like section 4.5.6 showed, each dataset that represents an artifact category gets its own classification pipeline consisting of general classification, artifact (group)-specific classification and source-specific classification. Because of that, five pipelines get defined in an orchestrator script that executes the pipelines sequentially. For every pipeline, it is important to know which column gets used for classification in which classifier. For all but the e-mail communication data pipeline, every classifier uses the same text artifact for every classifier in its pipeline:

- Code data: The actual code (column: code)
- Communication data (E-Mail): "subject" column for mail type classifier, "mail body" column for everything else.
- Communication data (Slack Messages): The individual message (column: message)
- Issue data: The issue title (column: issue_title)
- Version-Control data: The commit message (column: message)

Besides the composition of every created data pipeline, each step has to be configured to work with the given dataset. All steps need information on which column of the received data frame will be used for the text classification. Another configuration is the post-processing schema modifier. Because some column names in the source tables are not exact in their meaning, they get altered so that the final classified entries are easy to query.

When looking at the keyword-based classifiers, the keywords for each classifier

have to be configured. Because there is no further information about the issue or mail data available, the proposed keywords from the design description will be used. This does not affect the demonstration because the functionality of each classifier is still given. But it may impact further analysis based on the classification because the class labels may not be as precise as possible. This emphasizes the need for proper knowledge about the underlying software project to configure the classification system as accurately as possible.

For demonstrating the special character classifier, the chosen special characters are the Unicode emoji characters that were described in section 4.4.2.

5.3 Results of Classification

After executing the created pipelines, it can be shown that the classification system works as the design and the implementation details suggest. The data sink tables are created, and the columns for classification are filled. Appendix A shows the whole sample dataset and the assigned classifications as a final persisted data product in the database. If necessary, the text artifact was shortened. This was especially the case for the code data.

Within the code data, the general classification shows that all artifacts correctly have no searched special characters (emojis), include no Markdown media, and are a formal language. The only shortcoming can be seen with the "Inclusion of Markdown" classifier. Because, for example, Python comments have the same syntax as Markdown headlines, the classifier assumes that the text may be a Markdown document. It nonetheless works as designed. The source-dependent classifications all worked without fail. All artifacts were correctly classified as "code" and the languages were detected, too. Only one entry was classified as a "comment" because it predominantly consisted of comment lines. Listing 5.1 shows the code artifact that is used within the classification system itself.

```
1  # Util method for appending a string to the classification text
       to be persisted
2  # Dataframe: frame with the classification column
3  # index: Index showing the row of the dataframe for
       classification
4  # classification: string indicating the classification
       granularity
5  # new_classification: New classification string to append
6  # This method looks if a classification is already present
7  # Then it appends the new classification either
8  #       as a new string
9  #       with a comma onto the other string
10 # Works as a void method
11
```

```
12 def append_classification_string(dataframe, index, classification
   , new_classification):
13     old_classification = dataframe.at[index, classification]
14     separation_symbol = ", "
15 
16     if old_classification is None:
17         old_classification = ""
18         separation_symbol = ""
19 
20     dataframe.at[index, classification] = old_classification +
   separation_symbol + new_classification
```

Listing 5.1: Comment Code Artifact

The e-mail communication data was classified based on two different artifact text properties. For the general classification, the e-mail body was used. In general, no markdown media or special characters were detected. Similar to the code data, Markdown elements were found inside the text. Because of this, these artifacts were also declared formal languages. For the source independent classification, no e-mails apart from "normal" were found. None of the mail subject lines indicated that the mails are an answer or a forward of another mail. Therefore, the classification was correct.

The classification of Slack communication data also showed the workings of the proposed classification system. Each of the messages did not include any Markdown syntax for general classification and were all written in natural language. As a consequence, all artifacts were not classified as formal language. The Slack dataset was the only one for which a source-specific classification was applicable. The classifier correctly found all artifacts with Slack user handles and set the correct label. An example would be this message:

> *OK <@UUZKERH1S> Happy to hear that. Take care! Have a good day!*

For the issue data, the general classification was the same as for slack communication data. More interesting is the source-independent classification. For the "Inclusion of stacktraces" classifier, no artifact could be found that had any stack traces. The test data only provided the titles of issues but not their bodies. Their detection is, therefore, difficult. The principle based on regular expressions, though, has been shown to be effective by, for example, the Slack user handle classifier. Other than the stack traces, it was possible to detect special kinds of issues. In the sample data, bug issues were detected because they included the keyword "Fix". The whole issue title was: "*DEI Badging Application Bug Fix*". All other issues were classified as "unspecific issue".

The last dataset of Git version-control data holds no new information regarding general classification. All artifacts were correctly classified as being not formal

and having no special characters. For the source-independent classification, all artifacts were marked as human commits. Because there is no more information about the dataset, this classification can be considered correct. Regarding the type of commit, most of the artifacts were classified as "adaptive" as it is the default case for the classifier and many artifacts comply with the set keywords for that type (e.g. "Merge. . . "). Some artifacts were classified as corrective because they include the keyword "improve" hinting at a corrective change. When looking at the corrective message it can be seen that it starts with the adaptive keyword "merge":

> *Merge pull request #1433 from chaoss/libyear-worker-gsoc Fixed poetry parser and added poetry.lock parse*

Because the classifier was built in such a way that it prefers the label "corrective" over "adaptive" the "corrective" class was set. This is because the design stated that the classes have to be in a hierarchy that is achieved via the order of the keyword class checks (see Implementation section). The class "Corrective" is preferred in the implementation.

6 Analysis of Classifications

As an addition and answer to the third research question from the introduction, examples for the analysis of software artifacts based on the classification should be presented. These examples are all based on the classifications showcased in the design section of this thesis. Other examples with the designed or other classifications are possible.

The introduction of the thesis mentioned problems that come up with the implementation of inner source within organizations. These issues mainly concerned traceability and a lack of knowledge about the development process within an organization. Firstly, each of the classifications itself offers some sort of new knowledge that allows for a more in-depth analysis of the organization. One example for each of the artifact categories and the general classification should be given:

- **General Classification**: *How often are media inclusions part of software artifacts?* This analysis would be based on the "Inclusion of Media" classifier and would simply put the two classification labels into relation. Instead of taking all possible artifacts into account, features like the artifact category can be considered to increase the significance of the analysis. The analysis itself could be used to monitor the composition of documentation within an organization.
- **Build Data**: *How are software builds evolving over time?* With the classification of the build log, it would be possible to analyze it. Within the entire organization, the state and trend of software builds could be monitored, offering more transparency for controlling.
- **Code Data**: *What kind of code is how often submitted?* The "Code Type" classifier labeled code artifacts as "code" and "comment". This information describes what kind of code is submitted within an organization. That information could allow the assessment of productivity within the organization.
- **Communication Data**: *How much of e-mail traffic consists of original*

messages? The "Type of e-mail" classifier allows classifying e-mails into "original" or "normal e-mail", "forward", and "answer". With this information, the communication flow within an organization can be monitored. It can, for example, be seen if a lot of back-and-forth communication within the organization is handled via e-mail rather than other more efficient modes of communication.

- **Documentation Data**: *What kind of documentation is used?* One of the proposed classifiers for documentation data was the "Type of Documentation Form" classifier. With its capabilities, it would be possible to monitor the kind of documentation used and its frequency. This provides insight into the productivity of documentation creation and the usage of certain kinds of documentation, like diagrams or other inclusions of media, in the documentation text.
- **Issue Data**: *What is the share of bug tickets in comparison to other ones?* Classifying the "Type of Issue" allows for analysis of the share of, for example, bug tickets. This can be useful to track what kind of issues projects within the organization are handling.
- **Version-Control Data**: *What kind of the primary changes are done to the code base of a project?* By classifying the types of commits within the version-control data of a project, the kinds of primary changes can be detected. With this, an analysis of the state of a project is possible. Either it is in a state where a lot of adaptive and productive changes and additions are made, corrective fixes are applied, or perfective refactorings take place.

Other than analytics based on only one classification, multiple classification features can be combined to provide further insight into the software development process. Examples of that may be the following:

- *What type of commit do humans and bots usually commit?* This question combines the two presented version-control data classifiers and delivers information about the type of commit for a certain issuer. It can, for example, show what kinds of commits are automated via bots within the organization.
- *What kind of issues do usually include stacktraces?* This analysis also combines two classifiers of the same artifact category and creates an understanding of when stacktraces (or potentially other expressions) usually appear.

Other than just looking at the classifications, multi-feature analysis is also possible by taking meta-data into account. The described data pipeline was explicitly designed to support the usage and transport of the metadata available for each artifact. With these additional features, more analysis is possible. It is, for ex-

ample, possible to take time into account and draw certain conclusions from it. For example, it would be possible to see what type of code is submitted at what time.

All of these examples only show the surface of possible analysis with the help of the classification system. Together with more classifications and metadata, additional analysis is possible.

7 Evaluation

As part of the design science method, this chapter evaluates the results of this thesis. This evaluation consists of two parts. At first, the solution will be assessed regarding its capability to fulfill the objectives that are defined in chapter 3. Secondly, the limitations of the classification system and the solution in general will be pointed out. This helps to identify the potential for further research.

7.1 Evaluation of Defined Objectives

Within this thesis, four main objectives were defined based on the problem definition. It has to be examined whether the objectives were met. Each objective will be checked separately.

1. *The classification system is able to classify different kinds of software development artifacts.*

It can be said that this objective has been fulfilled. The objective can be split into two parts. At first, the classification system has to be able to classify software development artifacts. This was shown with the demonstration. With general, artifact-specific, and source-specific classifications, the system was able to label software artifacts with various classifications. The second part of the objective focuses on the ability to classify different kinds of artifacts. This, too, is part of the classification system. The system is designed in a way to allow for different data pipelines to cater to the various needs of diverse software artifacts. With these pipelines, it is possible to guarantee individual processing for different software artifacts with standalone pipeline steps, while making configuration easy through the modularity of each pipeline.

2. *The classification system is able to read data from various data sources and is not bound to a fixed set of sources.*

This objective is also fulfilled. The system was designed in such a way that it is not dependent on a fixed set of inputs. This can be seen in the design and the implementation. In the abstract data pipeline definition and the final design, it

can clearly be seen that the subject of data retrieval for different software artifacts is completely abstracted from the actual classification. Because of this, various data sources can be integrated into the system as long as they conform to the currently implemented data pipeline. How they conform is determined by the implementation and the form in which the data is presented. This thesis showed, as part of the demonstration, that collected software artifact data can come from special tools like GrimoireLab that help gather the diverse datasets. The final data format in the demonstration was a relational database, which allowed access within the implementation.

3. *The resulting classification results are usable within a wide range of assessment methods and analytics to support the inner source methodology.*

Chapter 6 showed how the presented classifications are usable in analytics and provide an assessment of the software artifacts. The chapter also demonstrated how these analytics can be used for monitoring and surveillance of classified software projects within an organization, which is an important goal in order to create benefits for the inner source methodology. But because this thesis does not deliver a thorough analysis based on the classification, it can only be concluded that the objective is partly fulfilled. Examples show the potential of the classification, but no final assessment can be made without further research.

4. *The classification system does not use machine learning techniques for artifact classification in order to assure repeatability.*

The last objective can also be verified. All classifiers were designed without any machine learning methods. This was done to assure reproducibility. All the methods and implementations for the classification of software artifacts were designed and programmed in a way that ensured the equivalent classification of the same text artifact every time. The classification is not dependent on any kind of training data set or something similar.

7.2 Limitations

Besides fulfilling the set objectives, certain limitations of this thesis can be formulated.

As mentioned in the problem identification section, the thesis focused on text-based classification. It was shown that classification can also be based on different metadata about the artifact, allowing for classifications that go beyond the analysis of text. As a consequence, the proposed classification could be expanded, delivering additional or refined classifications. This expansion could result in changes or additions to the presented demonstration, and therefore also change the perspective on possible analytics based on the classification system.

Another limitation lies within the demonstration dataset. While it was able to show the functionality of the classifier in a way that showed its functionality, the set still was limited and mostly from one single source project (GrimoireLab). To further verify the results of this thesis, bigger and more diverse datasets can be used to enhance the presented solution.

Similarly, the demonstration showed shortcomings of the implemented classifiers (Python code comments were detected as Markdown headings.) because of their generalized configuration. While the concept of the classification pipeline proofed effective, not every classifier is precise enough to generalize its effectiveness on other and more diverse datasets.

Furthermore, the analytics based on the proposed classifications only fulfill an exemplary role. This was already mentioned in the evaluation of the objectives. Because of this, the corresponding objective could only be partly verified. More research is necessary into the analytics of classified software artifacts to come to more substantial results in this domain.

8 Conclusions

The final chapter summarizes the results of this thesis and presents how future research can build upon them.

8.1 Summary

In the introduction, three central research questions were formulated. These questions specified the main issues that had to be worked on within the thesis. For the summarization, each question will be examined, and it will be shown how the thesis answered it.

1. *What are the objectives for a software development artifact classification system?*

The first part of the thesis and the design science methodology looked at the problem at hand and the final objectives that are implied by the problem identification. The problem identification consisted of two parts. At first, related work and tools were presented to show similar work and sources for inner source. The presented related work, on the one hand, showed a lot of domain-specific approaches, on the other hand, they proposed the not-wanted usage of machine learning techniques for classification. The related tools showed possible usable tools for the classification system. In the second part, the problem definition presented the different dimensions of the posted problem. These are the kinds of artifacts that have to be classified, the sources from which the artifacts come, and the final classes that are wanted in a classification system for those artifacts. An important finding was the focus on text-based classification.

With the identification of the problem, four objectives were formulated. These objectives all referenced an issue that was determined by the problem identification. All the objectives were properties the designed classification had to fulfill. This answered the first research question. The objectives themselves specified that the classification system has to be able to classify different kinds of artifacts, read from various data sources, create usable classifications, and work without using machine learning.

2. *How can such a classification system be conceptually designed and implemented?*

In the next part of the design science methodology, the thesis was concerned with the design and implementation of a classification system that fulfills the objectives that were set. For the design, a process was defined that at first described the starting situation. This included creating a hierarchy for the classification granularity into general, artifact-specific, and source-specific classifications. Several wanted classifications were presented.

In a first step, the classification system was designed as an abstract data pipeline consisting of data retrieval, preprocessing, classification, post-processing, and data writing. These abstract steps were then populated with concrete designs and implementation ideas. For later implementation, the classifiers were grouped into similar working processes. It was also shown that the modular design of pipeline steps allows for different pipelines for each artifact category. The section about the implementation showed how the design idea was worked out within the Python programming language environment.

At last, the demonstration showed the working of the design and the implementation by applying the created program. A sample dataset was used to test the classification system. The demonstration emphasized that the system worked as designed. Together with the evaluation of the objectives, the thesis proved that the system works sufficiently for the presented dataset, which shows that the proposed design answers the research question.

3. *What kind of analyses are possible with the developed classification system?*

The last research question was answered within the chapter about the analysis of classifications. Multiple examples were given for analytics that are now possible with the newly created classification system. They showed how classifications on their own or the combination of classifications with other classifications or metadata can create new knowledge. This can be done with single classifications or the combination of these with themselves or with metadata.

8.2 Future Work

Building upon this thesis, different types of future research can be done. As it was seen in section 7.2, limitations remain. These limitations can be used to expand on the present work. One important task would be the expansion of the classification system to include other factors than just the text of an artifact. This was described in chapter 2 and called "artifact-based" classification.

Another promising field for future work is analysis based on the classification of software artifacts. This thesis only provided examples for analytics to show the

applicability of the classification in the inner source context. Further work could analyze this topic more thoroughly.

Another part that profits from additional research is the extraction of data from the original data sources. Within the designed classification system, this step was heavily abstracted. More research can show how to use different data sources that could be integrated into the classification system pipeline.

Finally, the limitations mentioned the shortcomings of the demonstration dataset and the configuration of the classifiers. It is necessary to further refine the classification pipeline and its classifiers depending on other datasets. Future work should look into the refinement and application of the designed classification system with more advanced and complicated underlying software projects and data foundations.

Appendices

A Classified Software Text Artifacts (Demonstration)

On the following pages the tables with the demonstrated classifications are depicted. Each artifact category gets its own table. The left columns show the text artifact as a whole and if necessary in a shortened form. The right columns named "general classification", "source independent classification" and "source specific classification" state the set classes of the artifact.

A.1 Code Data

code	**general classification**	**source independent classification**
"public interface Weapon { void wield(); void swing(); ..."	"no_emoji, no_markdown, no_markdown_media, formal_language"	"code, language_java"
"import javax.sound.sampled.UnsupportedAudio FileException; ..."	"no_emoji, markdown, no_markdown_media, formal_language"	"code, language_java"
"public class ElfKingdomFactory implements KingdomFactory { ..."	"no_emoji, no_markdown, no_markdown_media, formal_language"	"code, language_java"
"@Slf4j public class Wizard { ..."	"no_emoji, no_markdown, no_markdown_media, formal_language"	"code, language_java"
"public class ClubbedTroll implements Troll { ..."	"no_emoji, no_markdown, no_markdown_media, formal_language"	"code,language_java"
"int main () { int x = 123; printf ("2 digits padding: %02d \n\n", x);..."	"no_emoji, no_markdown, no_markdown_media, formal_language"	"code, language_c"
"public class DragonSlayer { ..."	"no_emoji, no_markdown, no_markdown_media, formal_language"	"code, language_java"
"from pipeline.utils.classification_string_appender import append_classification_string ..."	"no_emoji, markdown, no_markdown_media, formal_language"	"code, language_python"

"from pipeline.utils.classification_string_appender import append_classification_string ..."	"no_emoji, no_markdown, no_markdown_media, formal_language"	"code, language_python"
"int main() { char str[] = "1234"; int x = atoi(str);..."	"no_emoji, no_markdown, no_markdown_media, formal_language"	"code, language_c"
"from pipeline.utils.classification_string_appender import append_classification_string ..."	"no_emoji, no_markdown, no_markdown_media, formal_language"	"code, language_python"
"public abstract class LetterComposite {..."	"no_emoji, no_markdown, no_markdown_media, formal_language"	"code, language_java"
"public class Messenger {..."	"no_emoji, no_markdown, no_markdown_media, formal_language"	"code, language_java"
"@Slf4j public class DwarvenTunnelDigger extends DwarvenMineWorker {..."	"no_emoji, no_markdown, no_markdown_media, formal_language"	"code, language_java"
"int main() { char char_array1[] = "45.82734"; float float_value = atof(char_array1);..."	"no_emoji, no_markdown, no_markdown_media, formal_language"	"code, language_c"
"int custom_abs(int n) { return (n < 0) ? -n : n; }..."	"no_emoji, no_markdown, no_markdown_media, formal_language"	"code, language_c"
"void print_string(const char *str) {..."	"no_emoji, no_markdown, no_markdown_media, formal_language"	"code, language_c"

"typedef enum {..."	"no_emoji, markdown, no_markdown_media, formal_language"	"code, language_c"
"int main() { int arr[] = { 10, 20, 30, 40, 50, 60 }; int size_arra = (arr, sizeof arr / sizeof *arr); ..."	"no_emoji, no_markdown, no_markdown_media, formal_language"	"code, language_c"
"# Util method for appending a string to the classification text to be persisted ..."	"no_emoji, markdown, no_markdown_media, formal_language"	"comment, language_python"

Table A1: Code Data classified

A.2 Communication data (E-Mail)

subject	**mail body**	**general classification**	**source independent classification**
"[CHAOSS] CHAOSSweekly (June 8 - 12, 2020)"	"CHAOSSweekly (Jun 8 - 12, 2020) Welcome Elizabeth Barron!"	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	normal_email
"[CHAOSS] Proposing GitBook as a platform for hosting Community-Handbook"	"Hi everyone, It sounds like there are no strong opinions"	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	normal_email
"[CHAOSS] Proposing GitBook as a platform for hosting Community Handbook"	"Thanks Sean, It?s a great idea to trial GitBook with a non-programmer."	"no_emoji, markdown, no_markdown_media, formal_language"	normal_email

[CHAOSS] Script for video about metric release	"Hi everyone, We discussed yesterday during the Community Meeting on June 23 that we"	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	normal_email
[CHAOSS] Doubt about the Elephant Factor definition and formula	"Thanks Alberto, The formula seems to always return 1/2 of total number of"	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	normal_email
[CHAOSS] CHAOSScast Episode 7 Is here!	"Dylan, thank you for publishing and sharing the latest CHAOSScast episode!"	"no_emoji, markdown, no_markdown_media, formal_language"	normal_email
[CHAOSS] Coding Period 1 Week-4 Update	"Hi Venu, Thank you for the update."	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	normal_email
[CHAOSS] GSOC update - Week 8	"Hi Sarit, Thanks for sharing your update."	"no_emoji, markdown, no_markdown_media, formal_language"	normal_email
"[CHAOSS] Piloting CHAOSS Community Report creation with Augur and GrimoireLab/Cauldron"	"Hi everyone, At today's community call, we"	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	normal_email
"[CHAOSS] CHAOSScast hits 1,000 downloads"	"Hi CHAOSS Community, Today, our community podcast, CHAOSScast, reached 1,000 downloads. This is"	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	normal_email
"[CHAOSS] Understanding Working Groups better - context for Community Handbook"	"Hi Jaskirat, I hope the input we provided today during the"	"no_emoji, markdown, no_markdown_media, formal_language"	normal_email

"[CHAOSS] Do we follow any community values - context for Community Handbook"	"Thanks Jaskirat, I agree with your suggestion to make the"	"no_emoji, markdown, no_markdown_media, formal_language"	normal_email
"[CHAOSS] Came across the new idea for proposing a metric and filtering it"	"The concern I have with using GitHub issues is that they are only visible within the respective repository."	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	normal_email
[CHAOSS] Outreachy Internship	"Hi Tola, Thanks also from me for all the things you have done for CHAOSS."	"no_emoji, markdown, no_markdown_media, formal_language"	normal_email
[CHAOSS] Path to Leadership - context for Community Handbook	"Hi Jaskirat,Thank you for this good framework."	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	normal_email
[CHAOSS] CHAOSScast Episode 18 Is Here!	"Thanks Dylan! I look forward to listening to this episode over the weekend."	"no_emoji, markdown, no_markdown_media, formal_language"	normal_email
[CHAOSS] GSoD: Week 5 progress of Jaskirat Singh	"Hi Jaskirat, Thanks for sharing and making great progress."	"no_emoji, markdown, no_markdown_media, formal_language"	normal_email
[Chaoss-Board] Community Bridge Funds	"Both times work for me. On Tue & Feb 4 & 2020 at 3:19 PM Michael Dolan <mdolan at linux-foundation.org> wrote:"	"no_emoji, markdown, no_markdown_media, formal_language"	normal_email
[Chaoss-Board] Draft Minutes	"+1 approved > On Apr 3, 2020, at 3:48 PM,"	"no_emoji, markdown, no_markdown_media, formal_language"	normal_email

Table A2: Mail Data Classified

A.3 Communication data (Slack Messages)

message	general classification	source specific classification
<http://meet.google.com/amn-gqio-mze|meet.google.com/amn-gqio-mze>	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	no_slack_user_handle
<http://meet.google.com/haq-mvim-ejr|meet.google.com/haq-mvim-ejr>	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	no_slack_user_handle
<http://meet.google.com/chm-hops-ckx|meet.google.com/chm-hops-ckx>	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	no_slack_user_handle
"I am excited to be here, how do I get started? I just read the docs for the review testing guide."	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	no_slack_user_handle
Hi	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	no_slack_user_handle
....except for the start of the pilot testing!!!	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	no_slack_user_handle
"Hello everyone :slightly_smiling_face: I got here after receiving an email invitation to participate in D&I Badging Pilot Testing. I am really excited about joining you all."	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	no_slack_user_handle

Do we have Badging Hour today?	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	no_slack_user_handle
OK <@UUZKERH1S> Happy to hear that. Take care! Have a good day!	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	slack_user_handle
Can anyone send me the link?	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	no_slack_user_handle
I am out at a farm right now! So I will be on later	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	no_slack_user_handle
for hosting the documents I made a small demo with Gitbook and here is the link.	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	no_slack_user_handle
"Gitbook is pretty good compared to Github Wiki. As from my personal experience I will prefer Gitbook for hosting the documentation as it will be beneficial even for writers, admins and users. The navigation is also so easy in Gitbook and the user don"t need prior experience and the UI is simple so that user can"t get bored."	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	no_slack_user_handle
"Hello everyone, I just made a small demo with Gitbook"	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	no_slack_user_handle

Thank you Matt!	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	no_slack_user_handle
The D&I Meeting tomorrow is canceled I. Just learned today; it is on the document. <@U011Z9MGK24> and I will be working together this Friday instead of meeting around that time. <@U011Z9ME9UY> <@UV1U5CVFZ> Feel free to set another time to meet or you could just hang out at the scheduled time. Thanks!	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	slack_user_handle
<https://meet.google.com/eot-jikt-for>	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	no_slack_user_handle
That link above is correct	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	no_slack_user_handle
+1	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	no_slack_user_handle

Table A3: Slack Communication Data classification

A.4 Issue Data

issue title	**general classification**	**source independent classification**
Update diversity-inclusion-workgroup-weekly-call.md	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"unspecific_issue, no_traceback_stacktrace"
Home Page Update Suggestions	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"unspecific_issue, no_traceback_stacktrace"
Home Page Update Suggestions	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"unspecific_issue, no_traceback_stacktrace"
Blog Post: The CHAOSS App Ecosystem WG releases Metrics for OSS Event Organizers	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"unspecific_issue, no_traceback_stacktrace"
DEI Badging Application Bug Fix	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"bug_issue, no_traceback_stacktrace"
DEI Badging Application Bug Fix	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"bug_issue, no_traceback_stacktrace"
Photo Album	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"unspecific_issue, no_traceback_stacktrace"
Photo Album	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"unspecific_issue, no_traceback_stacktrace"

Photo Album	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"unspecific_issue, no_traceback_stacktrace"
Review Website (Marketing Audit)	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"unspecific_issue, no_traceback_stacktrace"
Broken anchors on Metrics page	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"unspecific_issue, no_traceback_stacktrace"
Fix list formatting and header. Add intro para.	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"unspecific_issue, no_traceback_stacktrace"
Klumb patch 1	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"unspecific_issue, no_traceback_stacktrace"
Collaboration Platform Activity	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"unspecific_issue, no_traceback_stacktrace"
New Metric: Bot Activity	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"unspecific_issue, no_traceback_stacktrace"
Review - Focus area renaming	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"unspecific_issue, no_traceback_stacktrace"
Technical Fork: Continuous Metric Release Candidate	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"unspecific_issue, no_traceback_stacktrace"

Technical Fork: Continuous Metric Release Candidate	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"unspecific_issue, no_traceback_stacktrace"
GMD: first response to issue duration	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"unspecific_issue, no_traceback_stacktrace"
GMD: issue resolution efficiency - What are abandoned issues?	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"unspecific_issue, no_traceback_stacktrace"

Table A4: Issue Data classification

A.5 Version-Control Data (Git)

message	**general classification**	**source independent classification**
Adding numpy conversions on issue updates.	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"adaptive, human_commit"
Merge remote-tracking branch 'origin/cntrb-breadth-patch-3' into cntrb-breadth-patch-3	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"adaptive, human_commit"
6	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"adaptive, human_commit"
7	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"adaptive, human_commit"
8	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"adaptive, human_commit"
Data model documentation updates	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"adaptive, human_commit"
Data model release documentation updates.	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"adaptive, human_commit"

"Merge pull request #1420 from chaoss/docker-doc-update Made docker docs more current before I start making QoL improvements"	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"corrective, human_commit"
"Merge pull request #1433 from chaoss/libyear-worker-gsoc Fixed poetry parser and added poetry.lock parser"	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"corrective, human_commit"
"Create release-drafter.yml template: | ## What's Changed $CHANGES"	"no_emoji, markdown, no_markdown_media, formal_language"	"adaptive, human_commit"
"Merge pull request #1437 from chaoss/libyear-worker-gsoc Added Conda support for libyear Worker"	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"adaptive, human_commit"
dealt with one more UNICODE issues and added repo_id to message ref tables.	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"adaptive, human_commit"
"adding missing '.info' to logging in PR workerAdded response logging to rate limit logic Addressing common error with saltstack/salt repos (same location in both the PR and Issue workers."	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"adaptive, human_commit"
uncommented variable that was failing for saltstack/salt	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"adaptive, human_commit"
Added error logging to saltstack/salt area with s_buf_encoded variable printing the entire stream if there is a 'bulk_insert' error.	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"adaptive, human_commit"

Fixed pull request label logic.	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"corrective, human_commit"
logging update.	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"adaptive, human_commit"
indent!	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"adaptive, human_commit"
logging	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"adaptive, human_commit"
"Sample Documentation of Augur Implementation For @Georg to review and comment."	"no_emoji, no_markdown, no_markdown_media, not_formal_language"	"adaptive, human_commit"

Table A5: Version-Control Data classification

References

Alqahtani, S. S., & Rilling, J. (2017). An ontology-based approach to automate tagging of software artifacts. *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 169–174. https://doi.org/10.1109/ESEM.2017.25

Bacchelli, A., Dal Sasso, T., D'Ambros, M., & Lanza, M. (2012). Content classification of development emails. *2012 34th International Conference on Software Engineering (ICSE)*, 375–385. https://doi.org/10.1109/ICSE.2012.6227177

Balzert, H. (2009). *Lehrbuch der softwaretechnik. 1: Basiskonzepte und requirements-engineering / helmut balzert. unter mitw. von heide balzert* (H. Balzert, Ed.; 3. Aufl). Spektrum Akad. Verl.

Baysal, O., Holmes, R., & Godfrey, M. W. (2012). Mining usage data and development artifacts. *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, 98–107. https://doi.org/10.1109/MSR.2012.6224305

Bird, J., Menzies, T., & Zimmermann, T. (Eds.). (2015). *The art and science of analyzing software data* [OCLC: ocn909330141]. Morgan Kaufmann.

Buchner & Riehle. (2022). Calculating the costs of inner source collaboration by computing the time worked [Meeting Name: Hawaii International Conference on System Sciences]. In *Proceedings of the 55th hawaii international conference on system sciences (HICSS): January 4-7, 2022, hyatt regency maui, hawaii, USA*. University of Hawai'i at Manoa, Hamilton Library.

Capraro, M., & Riehle, D. (2016). Inner source definition, benefits, and challenges [Place: New York, NY, USA Publisher: Association for Computing Machinery]. *ACM Comput. Surv.*, *49*(4). https://doi.org/10.1145/2856821

Chaturvedi, K., Sing, V., & Singh, P. (2013). Tools in mining software repositories. *2013 13th International Conference on Computational Science and Its Applications*, 89–98. https://doi.org/10.1109/ICCSA.2013.22

Dueñas, S., Cosentino, V., Gonzalez-Barahona, J. M., del Castillo San Felix, A., Izquierdo-Cortazar, D., Cañas-Díaz, L., & Pérez García-Plaza, A. (2021). GrimoireLab: A toolset for software development analytics. *PeerJ Computer Science*, *7*, e601. https://doi.org/10.7717/peerj-cs.601

Edison, H., Carroll, N., Conboy, K., & Morgan, L. (2018). An investigation into inner source software development: Preliminary findings from a systematic literature review [event-place: Paris, France]. *Proceedings of the 14th International Symposium on Open Collaboration*. https://doi.org/10.1145/3233391.3233529

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. M. (1994). *Design patterns: Elements of reusable object-oriented software* (1st ed.). Addison-Wesley Professional.

Hindle, A., German, D. M., Godfrey, M. W., & Holt, R. C. (2009). Automatic classication of large changes into maintenance categories. *2009 IEEE 17th International Conference on Program Comprehension*, 30–39. https://doi.org/10.1109/ICPC.2009.5090025

Ma, Y., Fakhoury, S., Christensen, M., Arnaoudova, V., Zogaan, W., & Mirakhorli, M. (2018). Automatic classification of software artifacts in open-source applications [event-place: Gothenburg, Sweden]. *Proceedings of the 15th International Conference on Mining Software Repositories*, 414–425. https://doi.org/10.1145/3196398.3196446

Nazar, N., Hu, Y., & Jiang, H. (2016). Summarizing software artifacts: A literature review. *Journal of Computer Science and Technology*, *31*(5), 883–909. https://doi.org/10.1007/s11390-016-1671-1

Peffers, K., Tuunanen, T., Rothenberger, M., & Chatterjee, S. (2007). A design science research methodology for information systems research. *Journal of Management Information Systems*, *24*, 45–77.

Pfeiffer, R.-H. (2020). What constitutes software? an empirical, descriptive study of artifacts [event-place: Seoul, Republic of Korea]. *Proceedings of the 17th International Conference on Mining Software Repositories*, 481–491. https://doi.org/10.1145/3379597.3387442

Rani, P., Panichella, S., Leuenberger, M., Di Sorbo, A., & Nierstrasz, O. (2021). How to identify class comment types? a multi-language approach for class comment classification [Publisher: arXiv Version Number: 2]. https://doi.org/10.48550/ARXIV.2107.04521

Rasiel, E. M. (1999). *The McKinsey way: Using the techniques of the world's top strategic consultants to help you and your business*. McGraw-Hill.

Sarwar, M. U., Zafar, S., Mkaouer, M. W., Walia, G. S., & Malik, M. Z. (2020). Multi-label classification of commit messages using transfer learning. *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 37–42. https://doi.org/10.1109/ISSREW51248.2020.00034

Stol, K.-J., Babar, M. A., Avgeriou, P., & Fitzgerald, B. (2011). A comparative study of challenges in integrating open source software and inner source software. *Information and Software Technology*, *53*(12), 1319–1336. https://doi.org/10.1016/j.infsof.2011.06.007

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

Yusof, Y., & Rana, O. F. (2010). Classification of software artifacts based on structural information. In R. Setchi, I. Jordanov, R. J. Howlett & L. C. Jain (Eds.), *Knowledge-based and intelligent information and engineering systems* (pp. 546–555). Springer Berlin Heidelberg.