NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

GUILHERME FERREIRA LACÃO

BSc in Computer Science

MINING SOFTWARE MODEL REPOSITORIES

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon

October, 2023

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

MINING SOFTWARE MODEL REPOSITORIES

GUILHERME FERREIRA LACÃO

BSc in Computer Science

Adviser: Miguel Carlos Pacheco Afonso Goulão
Associate Professor, NOVA University Lisbon

Examination Committee

Chair: João Manuel dos Santos Lourenço
Associate Professor, NOVA University Lisbon

Rapporteur: Jácome Miguel Costa da Cunha
Associate Professor, University of Porto

Member: Miguel Carlos Pacheco Afonso Goulão
Associate Professor, NOVA University Lisbon

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
October, 2023

Mining Software Model Repositories

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.10.10) [Lou21].

Acknowledgements

Firstly, I would like to thank my adviser, Professor Miguel Goulão, for all the continuous support, guidance and availability that he demonstrated throughout the completion of this dissertation. His valuable suggestions and insights were crucial in shaping the research direction and refining the quality of this dissertation. My sincere thanks.

Then, I would like to thank the Department of Computer Science of NOVA School of Science and Technology for all the knowledge they have provided me over the last two years.

I would also like to thank all the colleagues and friends I made throughout my academic journey. We spent countless hours collaborating on projects, and I am deeply grateful for the great moments we shared and all their help and support.

Also, I would like to thank my friends outside the university. Their presence during the most challenging moments has been invaluable and deeply appreciated. Their continuous encouragement and making me believe in my abilities provided me with the support to overcome obstacles and finish this dissertation.

Finally, I would like to thank my family for their unwavering support and love. A special thanks to my mother, father, brother and grandmother. Without them, I could not be where I am now. I am deeply grateful for their patience, advice, and above all, their continuous encouragement throughout my whole academic path, and for making me believe that I can always achieve my goals.

Abstract

Modelling languages in software development are crucial for capturing requirements and representing software designs, architectures, and implementations. This dissertation focuses on UML class diagrams, a modelling language widely adopted in object-oriented software development.

The quality of UML class diagram models can significantly impact the quality of the system they represent. Defects present in these models can hinder stakeholder understanding, introduce unnecessary complexity, and propagate to the developed system, leading to increased costs. Therefore, understanding the most common defects present in these diagrams is crucial.

Further, with the growth of publicly available repositories, a wealth of valuable information, including UML class diagrams, is accessible. This presents an opportunity to study a large number of models extracted from these repositories.

In this dissertation, we present an automated evaluation tool to assess a dataset consisting of 103,103 UML class diagrams to identify the defects present in these diagrams. The creation of this dataset involved the development of a web scraping tool designed to extract UML class diagrams from public repository projects. The principles of the Physics of Notations proposed by Moody and the principles of diagram size and diagram flaws proposed by Störrle are incorporated into the automated evaluation tool to identify defects. This allowed us to analyse how UML class diagrams available in public repositories are built "in the wild", and to detect which are the most frequent violations of the modelling principles proposed by Moody and Störrle.

Keywords: UML Class Diagram, Model Quality Factors, Physics of Notations, Mining Repositories, Web Scraping, Modelling

Resumo

As linguagens de modelação no desenvolvimento de *software* são cruciais para capturar requisitos e na representação de *designs*, arquiteturas e implementações de *software*. A presente dissertação foca-se nos diagramas de classes UML, uma linguagem de modelação amplamente adotada no desenvolvimento de *software* orientado a objetos.

A qualidade dos modelos de diagrama de classes UML pode impactar significamente a qualidade do sistema que estes representam. Defeitos presentes nestes modelos podem dificultar a compreensão dos *stakeholders*, introduzir complexidade desnecessária e propagar-se para o sistema desenvolvido, levando ao aumento de custos. Portanto, compreender os defeitos mais comuns presentes nestes diagramas é crucial.

Além disso, com o crescimento de repositórios publicamente disponíveis, uma vasta quantidade de informações valiosas, incluindo diagramas de classes UML, está acessível. Oferecendo, deste modo, a oportunidade de estudar um grande número de modelos extraídos destes repositórios.

Nesta dissertação, é apresentada uma ferramenta de avaliação automatizada para avaliar um *dataset* composto por 103,103 diagramas de classes UML, a fim de identificar defeitos presentes nestes diagramas. A criação deste *dataset* envolveu o desenvolvimento de uma ferramenta de *web scraping* projetada para extrair diagramas de classes UML de projetos de repositórios públicos. Os princípios da *Physics of Notations* propostos por Moody e os princípios de *diagram size* e *diagram flaws* propostos por Störrle são incorporados na ferramenta de avaliação automatizada para identificar defeitos. Isto permitiu-nos analisar como os diagramas de classes UML disponíveis em repositórios públicos são construídos "*in the wild*", e detetar quais são as violações mais frequentes dos princípios de modelação propostos por Moody e Störrle.

Palavras-chave: Diagrama de Classes UML, Fatores de Qualidade de Modelos, *Physics of Notations*, Mineração de Repositórios, *Web Scraping*, Modelação

Contents

List of Figures

List of Tables

List of Listings

Acronyms

BPMN Business Process Model and Notation *(pp. 26, 32, 96)*

CNNs Convolution Neural Networks *(p. 12)*
CSV Comma-Separated Values *(pp. 51, 67)*
CVCSs Centralized Version Control Systems *(pp. 7, 8)*

DAG Directed Acyclic Graph *(p. 8)*
DOM Document Object Model *(pp. 13, 14)*
DVCS Decentralized Version Control System *(p. 8)*
DVCSs Decentralized Version Control Systems *(pp. 7, 8)*

HTQL Hyper Text Query Language *(p. 13)*
HTTP Hypertext Transfer Protocol *(pp. 13, 14, 33, 35, 37–39)*

IQR Interquartile Range *(pp. 79, 80)*

PoN Physics of Notations *(pp. 2, 3, 5, 14, 25, 28, 43, 47, 95, 96)*

UML Unified Modeling Language *(pp. viii, ix, 1–5, 11, 12, 25–33, 36, 39, 40, 42–44, 46, 50, 51, 53, 55, 56, 58, 59, 62, 63, 65, 69, 71–74, 90, 92, 95, 96)*

VCSs Version Control Systems *(pp. 6, 8)*

XMI XML Metadata Interchange *(pp. xii, 3, 27, 31–33, 39–42, 50–53, 55, 56, 69, 72, 95, 96)*

1 Introduction

This chapter introduces this dissertation by first providing a description and context of the problem, as well as its motivations. Then, it outlines the objectives and key contributions. Finally, it concludes with an overview of the structure of the document.

1.1 Context and Description

Software Engineering is a branch of Engineering that involves all aspects of developing and maintaining a software product. More concretely, it is the application and study of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software [BB16].

Models play a crucial role in software engineering, as they provide a means of abstracting the complexities of software systems and representing them clearly and concisely [Küh06]. Among the various types of models used in software engineering, UML models have become increasingly popular in recent years. UML is a standardized general-purpose modelling language consisting of a set of diagrams developed to help system and software developers specify, visualize, construct, and document the artefacts of software systems. UML plays a fundamental role in object-oriented software development and the software development process. Using the UML helps stakeholders communicate, explore potential designs, and validate the architectural design of the software. It comprises several diagrams, including class, sequence, activity, use case, and component diagrams [Pad12; Par].

Among these, this work will focus specifically on UML class diagrams, which are the most commonly used type of UML diagram. This is because much software is based on object-oriented programming, and UML class diagrams show the static structure of a system, including classes, their attributes and behaviours, and the relationships between each class [Luc].

The success of software development depends, in part, on UML class diagrams. When these diagrams contain defects, stakeholders may misinterpret them, resulting in the creation of a system that deviates from its intended purpose. Therefore, it is important to

know the what are most common defects present in these diagrams.

To collect these models, this work also focuses on repository mining. The widespread use of version control systems, like Git, to manage the development of software projects has resulted in a vast amount of publicly available repositories on platforms such as GitHub [Gitb; HSA20]. These repositories offer a wealth of information, including numerous versions and details of development over time. Researchers mine these repositories to gain insights from some of the valuable data they contain [KCM07]. Additionally, repositories can also contain other artefacts, such as models. However, some platform's projects can only be accessed through web browsers, as they do not provide APIs to fetch the data. This work focuses on a specific repository mining technique called web scraping. Web scraping helps to extract unstructured web data and transform it into structured data that can be saved in a central database [Khd21].

This work is also centred on evaluating the quality of UML class diagrams, with a specific focus on identifying defects present within these models. The evaluation will be performed using established principles such as the Physics of Notations (PoN) proposed by Moody [Moo09] and the principles of diagram size and diagram flaws proposed by Störrle [Stö16].

1.2 Motivation

The use of modelling languages in software development is essential for capturing requirements in the early stages of development, as well as for supporting the description of designs, software architecture, and even detailed implementations later in the process [BH00]. UML class diagrams, in particular, have become a widely used modelling language in object-oriented software development. Using this modelling language, the models created represent the structure and relationships of classes in a system.

Thus, the quality of these models plays a crucial role in determining the quality of the system it models. If the model has several defects, it may obstruct effective communication among software engineers and other stakeholders, leading to the introduction of unwanted complexity in the system. This, in turn, increases costs and reduces the overall quality of the system. Furthermore, if these models contain defects, stakeholders may misinterpret them, resulting in the creation of a system that deviates from its intended purpose.

Hence, the importance of analysing how software engineers use this modelling language, in practice, to identify typical defects introduced in the modelling process. This analysis is essential for addressing these defects and producing better-quality models.

The increasing adoption of version control systems and the consequent growth of publicly available repositories has resulted in a vast amount of valuable information stored in these repositories. This information not only includes code but also models, including UML class diagrams. Further, this offers an opportunity to explore various modelling techniques employed by diverse practitioners.

1.3 Objectives

This dissertation has three main objectives. First, create a web scraping application to collect UML class diagrams from online repositories. Second, create an automated evaluation tool to analyse and evaluate the collected dataset of UML class diagrams to identify the defects present in each diagram of the dataset. Third, analyse the results obtained from the automated evaluation tool and identify the most common defects present in these diagrams.

To identify defects, the automated evaluation tool employs a set of principles, which includes some of the principles of PoN, a framework to evaluate, compare and design cognitively effective visual notations. As well as, the principles of diagram size and diagram flaws created by Störrle, which take into account the number of elements that diagrams contain and various aspects within the diagrams that constitute flaws. The aim is to systematically evaluate the dataset using these principles to find the defects present in the models.

1.4 Key Contributions

An initial contribution is the development of a web scraping application to collect UML class diagrams from the GenMyModel platform. This web scraper browses each public project of the GenMyModel platform repository, checks whether the project contains a UML class diagram, and if so, downloads the XMI file of the project.

Another contribution is the collection of a dataset consisting of 103,103 XMI files, all containing a UML class diagram. This dataset was made publicly available due to the lack of datasets with a large number of diagrams of this type.

The first main contribution is the development of an automated evaluation tool to evaluate the collected dataset and identify the defects present in each model. This tool uses principles of the PoN, as well as the principles of diagram size and diagram flaws from [Stö16] to identify these defects.

Finally, the second main contribution is the analysis and discussion of the results obtained from the automated evaluation tool, which identified the most common defects present in the collected models.

1.5 Structure

The structure of this document is as follows:

- Chapter 2 - Background: presents the base concepts related to this dissertation, namely, models and metamodels, version control systems, mining software repositories, PoN, and quality factors of UML diagrams.

- Chapter 3 - Related Work: presents the relevant research related to this dissertation, highlighting the significant differences and contributions.
- Chapter 4 - Data Collection: presents the data collection process used to create the UML class diagrams dataset used in this dissertation, namely, the challenges encountered, the data source, relevant technologies, the web scraper solution, as well as the details of its implementation. Finally, it shows the results obtained with the web scraper.
- Chapter 5 - Principles for the Evaluation of UML Class Diagrams: describes the principles chosen to evaluate the collected UML class diagrams and provides examples of when principles are respected and not respected.
- Chapter 6 - Evaluation Process: presents an overview of the evaluation process conducted in this dissertation.
- Chapter 7 - Implementation: describes the implementation of the evaluation tool used to assess the collected UML class diagrams. It details the implementation of each principle used in the automated evaluation process and provides relevant code snippets.
- Chapter 8 - Results: presents and describes the results obtained from the execution of the automated evaluation tool. It also presents the general results derived from the analysis of the collected dataset. Lastly, it presents a critical discussion of the results.
- Chapter 9 - Conclusions: presents the conclusions of this dissertation, its contributions and future work.

2

Background

This chapter presents the fundamental concepts related to the topic of this dissertation, namely, models and metamodels, version control systems, mining software repositories, PoN, and quality factors of UML diagrams. The main objective is to provide an overview of the relevant concepts and tools that form the basis of this dissertation.

2.1 Models and Metamodels

A model is an abstraction (simplified representation) of a system that allows predictions or inferences [Küh06]. Also, a model needs to have the following three characteristics [Sta73]:

- **Mapping** - a model is always a representation (mapping) of an original.
- **Reduction** - a model does not capture all properties of an original, only those that the modeller considers relevant.
- **Pragmatism** - a model must be usable in place of an original for some specific purpose and a within a specific time frame.

Steinmüller asserts that a model is information [Ste93] (cited in [Küh06]):

- on something (content, meaning),
- created by someone (sender),
- for somebody (receiver),
- for some purpose (usage context).

Kühne classifies models as [Küh06]:

- **Token models** - capture unique aspects of the elements in a system.
- **Type models** - capture universal aspects of the elements in a system.

A metamodel is a model of models, serving as a template for defining the concepts and relationships that can exist in other models. It outlines the types of features that can be used, the potential relationships between those features, and the constraints that must be satisfied for the overall validity and consistency of the modelled system [Küh06; MZZ06].

Figure 2.1 illustrates an example of a model of a real-world system through a token model, which captures singular aspects of the elements in the system, and its corresponding metamodel, which is a type model that captures universal aspects of the elements in the system. This highlights the distinction between a model, an abstraction of a real-world system created to make predictions or inferences, and a metamodel, which outlines the concepts and relationships that can exist in other models.

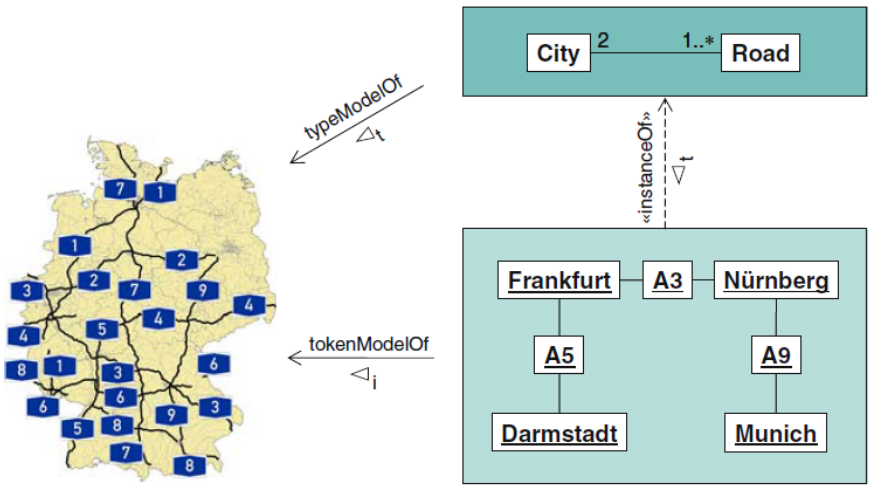


Figure 2.1: An example of model of a real-world system and its corresponding metamodel, adapted from [Küh06].

2.2 Version Control Systems

In software development, it can be challenging to coordinate and manage not only a large number of software developers working simultaneously but also the ever-growing software projects. Hence the need for Version Control Systems (VCSs). VCSs are systems that manage the development of evolving projects. That is, they keep up with all changes that software developers make [AS09; ZND18]. In particular, VCSs use snapshots to track changes to a folder and its contents, with each snapshot capturing the entire state of files/folders. They also save metadata associated with each snapshot [Ath].

VCSs are useful when working on an individual project or with other developers, as they keep a history of all the code versions and track who made the changes and to which artefacts. They also make it simpler for developers to work collaboratively and concurrently. As a result, the development process becomes easier and faster [ZND18].

The two most popular types of VCSs are [Gitg]:

- Centralized Version Control Systems (CVCSs) [AS09; Muş+14] store all the development history on a central server. Therefore, all developers work against the central repository through a point-in-time snapshot.

  In addition, CVCSs support *branches* for the parallel development of a repository. However, to perform operations that require access to history that is not available locally, they must be executed on the server, for example, the *merge* operation.

  The most widely used CVCSs are TortoiseSVN [Tor] and Subversion (SVN) [Apa; Sli].

- Decentralized Version Control Systems (DVCSs) [AS09; Bir+09; Muş+14] store all development history as a local repository. Thus, there is no centralized repository, with each developer maintaining their own. Also, in DVCSs, most operations are performed locally, except for synchronizing with another repository.

  A side benefit of DVCSs is that they inherently lead to the replication of source code across several places, reducing the risk of some disaster scenarios.

  The most widely used DVCSs are Git [Gita] and Microsoft Azure DevOps Server [Azu; Sli].

Over the years, the number of software projects using DVCSs has grown, and this trend appears to continue [Rea]. Consequently, software projects are transitioning, or are posing to transition, from CVCSs to DVCSs [Rea]. The reasons for this transition are the following [AS09; Gite; Muş+14]:

- **Fast merging** - DVCSs do not require remote server communication, allowing for quick and efficient merging of code, which is a common issue with CVCSs.

- **The ability to work offline** - CVCSs require the developer to connect to the central server before editing a file for the first time, which makes working offline a challenge. In contrast, DVCSs only require the developer to interact with the server when he needs to add changes to or retrieve new changes from the server.

- **Reliable backup copies** - CVCSs store all development history on a centralized server, which is a single point of failure in the event of a disaster. DVCSs, on the other hand, remove the dependency on a single backup, as each repository clone is essentially an offsite backup.

- **Improved support for experimental changes** - DVCSs cheap branches allow developers to make local commits to snapshot a work-in-progress, which is often desirable before starting some experimental work with an unknown outcome.

Despite the benefits of DVCSs over CVCSs, CVCSs have the following advantages [Gitf]:

- **Efficient handling of binary files** - Binary files present a storage challenge. CVCSs offer an efficient solution, allowing the retrieval of specific lines of code without storing the complete history locally. In contrast, the decentralized nature of DVCSs requires downloading the entire project.
- **Full project visibility** - CVCSs provide a centralized location for code management, allowing full visibility and understanding of the state of a project. This is achieved by accessing and tracking changes made to code on a central server. On the other hand, several dispersed repositories in DVCSs can make it difficult to understand and visualize the state of a project.

2.2.1 Git

Git is an open-source Decentralized Version Control System (DVCS) designed to handle both small and large projects quickly and effectively [CS14].

Despite having a user interface similar to others VCSs, Git stores and thinks about data differently. Other VCSs (such as CVS and Subversion) store information as a list of file-based changes, which means that they consider the information stored as a set of files and the changes made to each file over time. However, Git does not conceptualize or store its data in this way [CS14]. Instead, Git views its data as a series of snapshots of a group of files and directories contained in a top-level directory. Therefore, each time the project's state is changed, Git takes a picture of what all the files look like at that point in time and stores a reference to that snapshot. In Git nomenclature, a file is called a *blob*, and a directory is called a *tree*. A *tree* can contain both *blobs* and *trees* (meaning that a directory can have files and directories). So, a snapshot is the top-level *tree* that Git tracks [Ath; CS14].

In Git, the history is a sequence of snapshots. However, it is not a linear sequence. Instead, it is a Directed Acyclic Graph (DAG) of snapshots. This means that a snapshot has a set of parents (the snapshots that preceded it) rather than a single parent, as a snapshot can descend from multiple parents, for example, due to combining (*merge*) two parallel developmental branches [Ath].

In more detail, Listing 2.1 shows the pseudocode of Git's data model.

Listing 2.1: Pseudocode of Git's data model, adapted from [Ath].

```
type blob = array<byte>

type tree = map<string, tree | blob>

type commit = struct {
    parents: array<commit>
    author: string
    message: string
    snapshot: tree
}
```

A *blob* is a collection of bytes represented as an array of bytes. A *tree* (directory) is a mapping from the directory name to its contents, which is either a *sub-tree* (sub-directory) or a *blob* (file). Finally, a *commit* is a data structure that contains the following information [Ath]:

- **Parents** - the *commits* that precede the current *commit*, represented as an array of *commits*.
- **Author** - the author of the *commit*, represented as a string.
- **Message** - the description of the *commit*, represented as a string.
- **Snapshot** - the actual contents of a *commit*, which is the top-level *tree* corresponding to the *commit*.

Listing 2.2. shows the pseudocode of Git's data store. Git defines an *object* as a *blob*, *tree* or *commit*. Furthermore, Git is a content-addressable filesystem, which means it keeps a set of *objects* on disk as a simple key-value data store. So, to store a given *object*, Git creates a unique key and inserts the key and the *object* into the *objects* key-value data store [Ath; CS14]. In particular, computing the unique key for an *object* is done using the *SHA-1* cryptographic hash function, which transforms a given input (of any size) into a fixed-length output hash value of 160 bits [Sta11]. Also, a particular stored *object* can be loaded using its unique key.

Listing 2.2: Pseudocode of Git's data store, adapted from [Ath].

```
type object = blob | tree | commit

objects = map<string, object>

def store(object):
    id = sha1(object)
    objects[id] = object

def load(id):
    return objects[id]
}
```

Considering that *SHA-1* hashes are used to identify snapshots, this poses a problem as it is difficult for humans to remember strings of 40 hexadecimal characters. So, in addition to maintaining a set of *objects*, Git also contains a set of *references*. *References* are human-readable names for *SHA-1* hashes that are pointers to *commits*. Thus, allowing Git to refer to a specific snapshot using a human-readable name rather than a hexadecimal string. Listing 2.3 shows the pseudocode of *references* [Ath].

Listing 2.3: Pseudocode of Git's *references*, adapted from [Ath].

```
references = map<string, string>
```

Finally, a Git repository can be defined as a set of *objects* and *references*.

2.3 Mining Software Repositories

2.3.1 Mining Code Software Repositories

Mining software repositories is the field where relevant data from software repositories is extracted, processed and analyzed. In recent years, this field has seen tremendous growth due to the large amount of data made publicly available in software repositories hosted on platforms such as GitHub [Gitb], GitLab [Gitd] and SourceForge [Soub] [Has08; HSA20].

These repositories contain a plethora of information that offers insights into the development of software systems over time. This information can represent numerous versions with years worth of development details. Individual system versions, changes, and metadata (about the changes) are all part of this information. Therefore, researchers mine these repositories to gain insights from some of this valuable data. Consequently, it enables them to identify repository patterns and relationships related to code ownership, software evolution, and more [KCM07].

The process of mining software repositories involves several stages, including extracting and selecting data, data preparation and cleansing, and data analysis. Regarding data extraction, the most significant data sources are open-source software repositories, where GitHub typically is used due to the considerable volume of repositories it hosts [HSA20].

Furthermore, there are several ways of extracting data from software repositories, including [HSA20]:

- **GHTorrent** [Gou13; GS12] - This project is a service that collects event streams and data from GitHub and makes this data available to the community in the form of incremental data dumps via a peer-to-peer protocol. It consists of two databases: one for mirroring the event streams of the GitHub repositories (such as `pull`, `push` and `fork` requests) and the other for the repository's metadata. However, GHTorrent does not provide the repository's structure or contents.

- **GitHub API** [Gitc] - GitHub provides a public REST API that enables the extraction of repository contents, metadata (such as information about the repository, commits and collaborators), and structure. The API can only be used with authentication, with each account having a limit of 15000 requests per hour. To ease this restriction, multiple accounts can be used simultaneously.

- **Google BigQuery** [Goo] - This service provides various datasets with data from GitHub repositories. Using the Google BigQuery service allows for querying massive datasets with SQL queries in a scalable way. However, this service has rate limits. Concretely, 1TB per month of query data processed is free to access.

2.3.2 Mining Software Model Repositories

The mining software repositories field has primarily focused on the source code of programming languages. However, repositories can also contain other artefacts, such as

models. Thus, researchers can take this process further by mining software repositories to find specific types of models as well as the related source code and development metadata [Heb+16].

There is a wide range of mining techniques. However, since the methods used in model mining share several commonalities, they can be assembled as an all-encompassing method. Figure 2.2 illustrates this overall mining process. This method comprises the following steps [Heb+16; HSA20; Ho-+17; Rob+17]:

1. **Repository Selection** - To obtain a comprehensive and up-to-date list of software repositories hosted on GitHub, GHTorrent is utilized instead of directly accessing the GitHub platform. This approach is preferred due to the advantages offered by GHTorrent, including its lightweight and easy-to-use nature, its ability to foster replicability, and its flexibility for researchers [Gou+14]. The most recent database dump from GHTorrent is downloaded to provide a list of millions of repositories, which excludes those that have been forked or deleted. Note that GHTorrent does not provide information about the files in the repositories.

2. **Data Extraction** - This step is divided into three sub-steps:

   a) **Repository Files Extraction** - To overcome the limitation of GHTorrent, the GitHub API is used to fetch the list of files from the previously identified repositories. The result is a JSON file per repository with information about the files stored on the *master* branch. However, downloading all the JSON files is incredibly slow as the GitHub API has a limit of 15000 requests per hour. To reduce this time, multiple GitHub accounts can be used to download the JSON files in parallel;

   b) **UML Files Identification** - To identify UML models, multiple file formats must be considered. Word documents (.doc(x)), Portable Document Format documents (.pdf), and PowerPoint slides (.ppt(x)) are typically excluded, as there is no current automated method for extracting UML models from these formats. Therefore, the file extensions of interest are the following:

      i. Standard formats: ".uml" and ".xmi".

      ii. Image formats: ".xml", ".bmp", ".jpg", ".jpeg", ".gif", ".png" and ".svg".

      Additionally, files with other extensions can be kept based on specific filenames set by the researchers. It is important to note that this step only identifies files that may contain UML, so they may or may not have UML;

   c) **UML Files Extraction** - Finally, the identified files are automatically downloaded.

3. **Filtering and Cleansing** - In this step, the identified files are checked to see if they contain UML diagrams. This step is divided into two sub-steps:

a) **UML Images Verification** - Before the verification process begins, images that cannot be opened, do not meet specified dimensions (set by the researchers), or are duplicates are removed. The remaining images are then classified as UML or non-UML images. For this classification, different studies used different approaches: some using Machine Learning algorithms and others using Deep Learning algorithms, such as Convolution Neural Networks (CNNs). The algorithms used in these studies and their corresponding accuracy are summarized in Table 2.1. Additionally, manual verification can be performed on the set of images to detect false positives and negatives.

Table 2.1: Classification algorithms and corresponding accuracy from different studies, adapted from [Gos+21].

Authors	Number of images	Algorithm	Accuracy
B. Karasneh et al. [KC13]	200	N/A	89%
T. Ho-Quang et al. [Ho-+14]	1300	SVM	91.2%
		Random Forest	90.7-93.2%
R. Hebig et al. [Heb+16]	19 506	Random Forest	98.6%
V. Mareno et al. [Mor+16]	18 899	image-to-model (i2m)	94.4%
J. Ott et al. [OAL19]	11 319	Low-short learning with CNNs	92%
B. Gosala et al. [Gos+21]	3282	CNNs without regularization	85.41%
		CNNs with regularization	86.63%

b) **UML Files (".uml" and ".xmi") Verification** - The ".uml" files are typically all considered, as the files with this extension certainly use the UML notation. Concerning the ".xmi" files, they contain a schema that specifies the format of the content. Therefore, by parsing them for their schema references, it is possible to verify whether they include UML models. Lastly, the hash values of these files are calculated to detect duplicates.

4. **Metadata Extraction** - This step involves downloading all the repositories that contain at least one UML file identified. The `perceval` tool [Due+18] is then used to extract the metadata from these repositories. Note that this tool can retrieve metadata from the repositories in parallel.

5. **Analysis** - In the final step, the analysis of the UML files and the metadata from the repositories is performed, typically through statistical and computational methods.

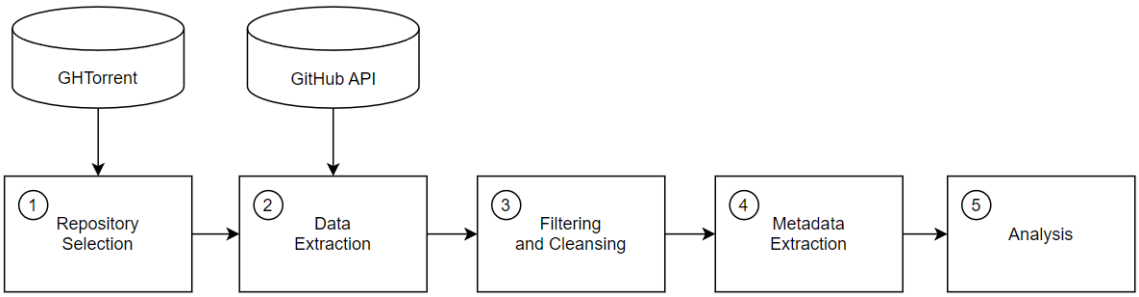


Figure 2.2: Overall mining process, adapted from [HSA20; Ho-+17].

2.3.3 Web Scraping

The World Wide Web consists of large amounts of unstructured data in the form of text, images, audio and video contained within websites, which can only be accessed through web browsers. Extracting this data from websites involves manually copying and pasting the desired web page content onto a computer's hard drive, which can be a tedious and time-consuming process [SMP19]. Additionally, some websites provide APIs to interact with or fetch data. However, it is not uncommon for APIs not to meet the user's needs [Per19]. We need a faster and more effective approach to collect large amounts of data from multiple websites. Web scraping offers just that [Khd21].

Web scraping is a technique for extracting unstructured web data and transforming it into structured data that can be saved and analyzed in a central database or spreadsheet. This technique enables the retrieval of large amounts of data within a short amount of time [Khd21].

There are several web scraping techniques, including [Khd21; Sir+15]:

- **Traditional copy-and-paste** - This technique consists of human manual examination and copy-and-paste method, which can occasionally be practical and effective. However, it is an error-prone, tiring and tedious technique when analyzing large numbers of datasets.
- **Text grabbing and regular expressions** - This technique is based on the UNIX command or regular expression-matching facilities of programming languages and is a simple approach to extracting data from web pages.
- **Hypertext Transfer Protocol (HTTP) programming** - This technique uses socket programming, making HTTP requests to a remote web server to extract data from static and dynamic web pages.
- **HTML parsing** - Semi-structured data query languages, such as XQuery and Hyper Text Query Language (HTQL) can be used to parse HTML pages and to retrieve and transform page content.
- **Document Object Model (DOM) parsing** - By embedding a web browser, such as Chrome or Mozilla browser control, programs can retrieve the dynamic content

generated by client-side scripts. These browser controls also parse web pages into a DOM tree, based on which programs can retrieve parts of the pages.

- **Web scraping software** - Several software tools can provide customized web scraping solutions. This software can automatically identify the data structure of a page and provide a recording interface that eliminates the need to write web scraping code. Additionally, some of these software tools provide scripting functions that can be used to extract and transform content, as well as database interfaces that can store the extracted data in local databases.
- **Computer vision web-page analysers** - This technique uses machine learning and computer vision to identify and extract useful information from web pages by scanning pages, mimicking what a human would do.

The web scraping process comprises three main phases, as illustrated in Figure 2.3, which are [Khd21; Per19]:

- **Fetching phase** - The desired website with the relevant data must be accessed using the HTTP protocol. This protocol is used to send requests and receive responses from the web server and is the same method used by web browsers to access web page content. In this phase, libraries such as `curl` [cur] and `wget` [Pro] can be used to send an HTTP `GET` request to the desired location (URL) and get the HTML document as a response.
- **Extraction phase** - After fetching the HTML document, this phase involves extracting the relevant data from the HTML document using technologies such as regular expressions, HTML parsing libraries, or XPath queries.
- **Transformation phase** - The remaining data of interest can be transformed into a structured version for storage or presentation.

2.4 Physics of Notations

The PoN is a design theory that focuses on the perceptual properties of visual notations. It defines a theory of how visual notations communicate, called Descriptive Theory (subsection 2.4.1), and based on this, a set of nine principles for evaluating, comparing, and constructing visual notations, called Prescriptive Theory (subsection 2.4.2) [Moo09].

2.4.1 Descriptive Theory

This subsection presents a theory of how visual notations communicate. Understanding how and why visual notations communicate is crucial in enhancing their ability to communicate [Moo09]. Figure 2.4 illustrates this communication model based on the theory of communication [Sha48].

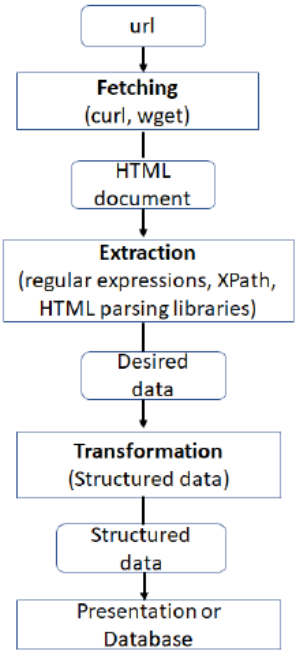


Figure 2.3: Web scraping process, adapted from [Khd21].

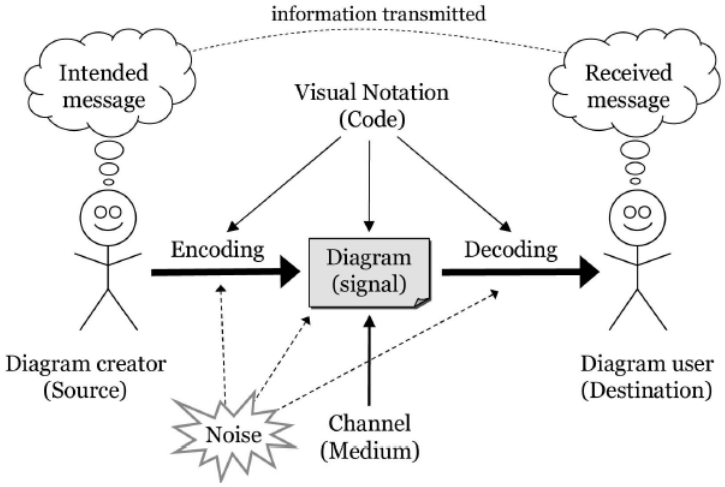


Figure 2.4: Theory of communication, adapted from [Moo09].

In the figure, the diagram creator (sender) encodes information (message) in the form of a diagram (signal), and the diagram user (receiver) decodes this signal. The diagram is encoded using a visual notation (code), which defines a set of conventions that both parties understand. The medium (channel) is the physical form in which the diagram is presented, for example, paper, whiteboard, or computer screen. Noise represents random variation in the signal which can interfere with communication. Thus, the effectiveness of communication is measured by the match between the intended message and the received message (information transmitted) [Moo09].

In this theory, communication involves two essential processes [Moo09]:

- **Encoding** - explores the available options for encoding information in visual form. Defines the design space, which refers to the set of possible graphic encodings for a given message.

  To graphically encode information, Bertin delineated eight visual variables (Figure 2.5) in his *Semiology of Graphics* work [Ber83]. These are divided into two categories: planar variables and retinal variables. The planar variables represent the two spatial dimensions: horizontal and vertical position. The retinal variables represent the features of the retinal image: shape, size, colour, brightness, orientation and texture.

  The visual variables define a set of atomic building blocks that can be used to build any visual representation. Visual variables thus delineate the dimensions of the graphic design space. Moreover, by combining the variables, an unlimited number of graphic symbols can be created.

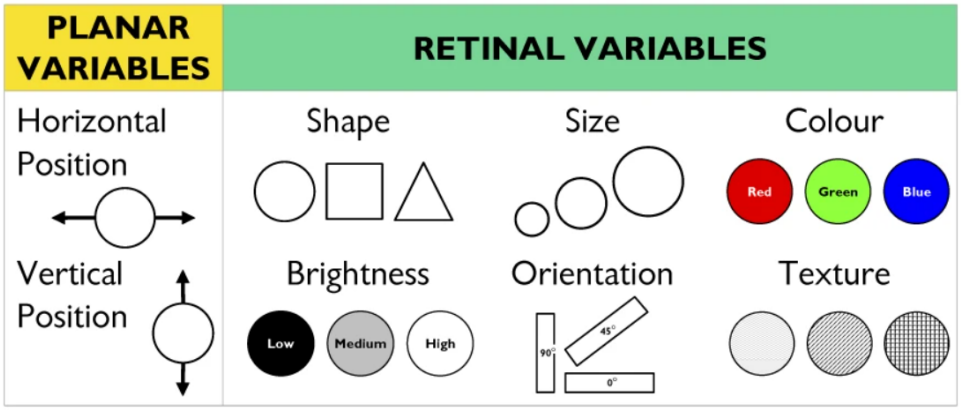


Figure 2.5: Visual variables, adapted from [Moo09].

- **Decoding** - explores how visual notations are processed by the human mind. Defines the solution space, which refers to the principles of human information processing that provide the basis for choosing among the unlimited number of possibilities in the design space.

  Figure 2.6 illustrates a model of human graphical information processing, which is divided into two phases: perceptual processing (seeing) and cognitive processing (understanding). Perceptual processes are automatic, very fast, and mostly executed in parallel. In contrast, cognitive processes operate under the conscious control of attention and are relatively slow, effortful, and sequential.

  Perceptual processing consists of two steps: perceptual discrimination and perceptual configuration. In the perceptual discrimination step, retinal image features, such as colour and shape, are detected by specialized feature detectors. Based on this, the diagram is parsed into its constituent elements and separated from the background. In the perceptual configuration step, structure and relationships between diagram elements are inferred based on their visual characteristics.

The transition between the perceptual processing phase and the cognitive processing phase occurs when all or part of the perceptually processed image is brought into working memory under the conscious control of attention.

Cognitive processing consists of two parts: working memory and long-term memory. Working memory is a temporary storage area used for active processing that reflects the current focus of attention. Its characterized by limited capacity and duration. Long-term memory is a permanent storage area with unlimited capacity and duration, but is relatively slow.

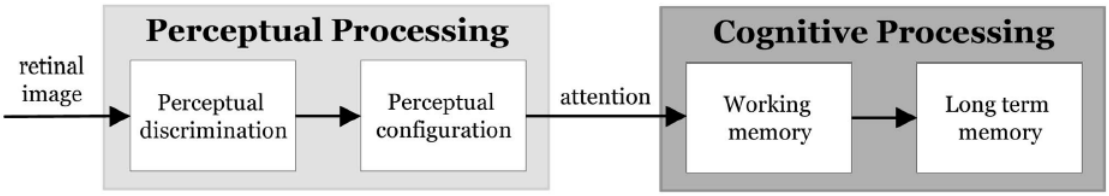


Figure 2.6: The solution space, adapted from [Moo09].

2.4.2 Prescriptive Theory

This subsection presents a set of nine principles for designing cognitively effective visual notations.

2.4.2.1 Semiotic Clarity

There should be a one-to-one correspondence between the concepts and the graphical symbols. When this correspondence is not present, it can lead to the following anomalies (Figure 2.7) [Moo09]:

- **Symbol redundancy** - when multiple symbols represent the same concept.
- **Symbol overload** - when the same symbol represents different concepts.
- **Symbol excess** - when a symbol does not represent any concept.
- **Symbol deficit** - when concepts are not represented by any symbol.

2.4.2.2 Perceptual Discriminability

The ease and accuracy with which graphical symbols can be differentiated from each other [Moo09]. Accurate discrimination between symbols is a prerequisite for the accurate interpretation of diagrams [Win90].

This principle encompasses the following concepts [Moo09]:

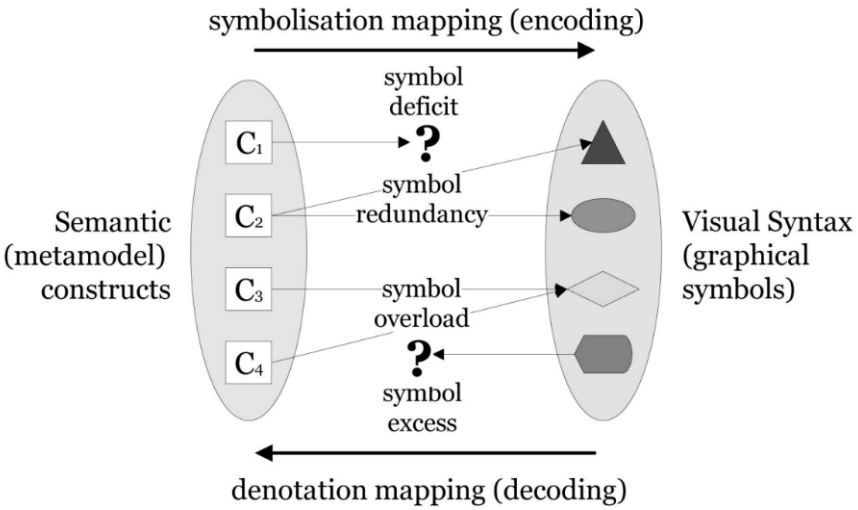


Figure 2.7: Principle of Semiotic Clarity, adapted from [Moo09].

- **Visual Distance** - discriminability is primarily determined by the visual distance between symbols, which is measured by the number of visual variables in which they differ and the size of these differences. Typically, the greater the visual distance between symbols, the faster and more accurately they will be recognized [Win93]. If the differences are too subtle, errors in interpretation may result, especially for novices [BJ99].

- **The Primacy of Shape** - shape, which serves as the primary basis for identifying objects in the real world, plays a crucial role among all visual factors in differentiating between symbols. For this reason, the shape should be used as the primary visual variable for distinguishing between different semantic constructs.

- **Redundant Coding** - multiple visual variables can be used to increase the visual distance between symbols.

- **Perceptual Popout** - visual elements with unique values for at least one visual variable can be detected pre-attentively and in parallel across the visual field [Qui03; TG80].

- **Textual Differentiation** - several notations use text and typographic characteristics (bold, italics, and underlining) to distinguish between symbols.

2.4.2.3 Semantic Transparency

Use visual representations whose appearance suggests their meaning. Thus, this principle emphasizes that symbols must provide clues to their meaning [Moo09].

Semantically transparent representations reduce cognitive load because they use built-in mnemonics, so their meaning can be perceived directly or easily learned [Pet95].

Semantic transparency can be categorized into four levels of transparency (Figure 2.8) [Moo09]:

- **Semantically immediate** - a novice reader can infer the symbol's meaning just by its appearance.
- **Semantically translucent** - symbols provide a cue to their meaning but require some explanation.
- **Semantically opaque** - no relationship between the symbol's appearance and meaning.
- **Semantically perverse** - the symbol's appearance causes a novice reader to infer a different or opposite meaning.

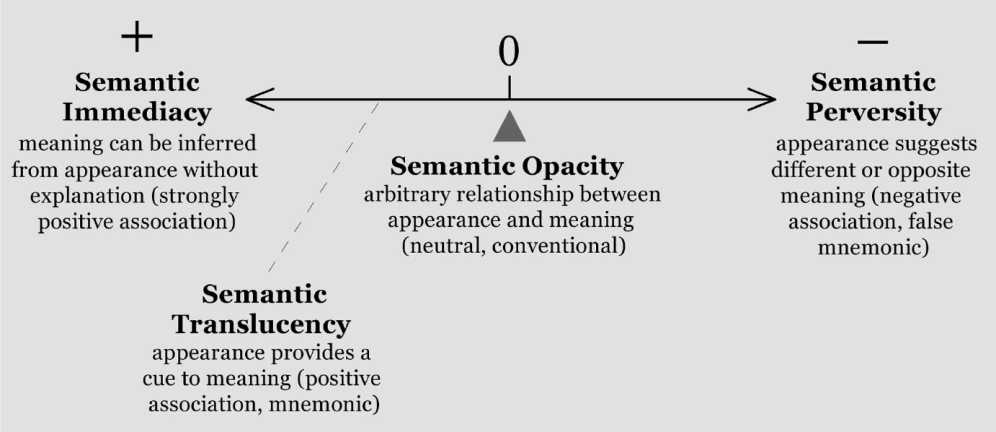


Figure 2.8: Levels of semantic transparency, adapted from [Moo09].

2.4.2.4 Complexity Management

Include explicit mechanisms for dealing with complexity. Therefore, this principle highlights the ability of a visual notation to represent information without overloading the human mind [Moo09].

In this context, "complexity" refers to diagrammatic complexity, which is measured by the number of elements in a diagram. Complexity has a big impact on cognitive effectiveness as the amount of information that can be effectively conveyed in a diagram is limited by [Moo09]:

- **Perceptual limits** - the ability to discriminate between diagram elements increases with diagram size [Pat03].
- **Cognitive limits** - the number of diagram elements that can be comprehended at a time is limited by working-memory capacity. When this is exceeded, a state of cognitive overload occurs [Mil56].

To effectively represent complex situations, visual notations must provide the following mechanisms [Moo09]:

- **Modularization** - involves dividing systems into smaller parts or subsystems. Thus, notations should provide the ability to break large diagrams into perceptually and cognitively manageable parts to avoid overloading the human mind. Cognitive load theory shows that reducing the amount of information presented at a time improves the speed and accuracy of comprehension and facilitates a deep understanding of information content [MM03].

- **Hierarchy (levels of abstraction)** - consists of decomposing systems into different levels of detail, with complexity manageable at each level [FC93].

2.4.2.5 Cognitive Integration

Include explicit mechanisms to support the integration of information from different diagrams. Note that this principle is only applicable when multiple diagrams are used to represent a system [Moo09]. This principle is significant since representing systems with multiple diagrams places additional cognitive demands on the reader, who must mentally integrate information from various diagrams while keeping track of where they are [Sia04].

The following explicit mechanisms must be present in multi-diagram representations for them to be cognitively effective (Figure 2.9) [Moo09]:

- **Conceptual integration** - mechanisms to help the reader assemble information from separate diagrams into a coherent mental representation of the system. Two crucial mechanisms are a summary diagram and contextualization. A summary diagram provides a view of the system as a whole, acting as an overall cognitive map into which information from individual diagrams can be assembled [KHH00; RA16]. Contextualization is a technique used in information visualization where the part of a system of current interest (focus) is displayed in the context of the system as a whole [LR96; Tur+04].

- **Perceptual integration** - perceptual cues to simplify navigation and transitions between diagrams.

2.4.2.6 Visual Expressiveness

Use the full range and capacities of visual variables. Thus, using a variety of visual variables results in a perceptually enriched representation that exploits multiple visual communication channels and maximizes computational offloading [Moo09].

Visual variables are classified into two subsets [Moo09]:

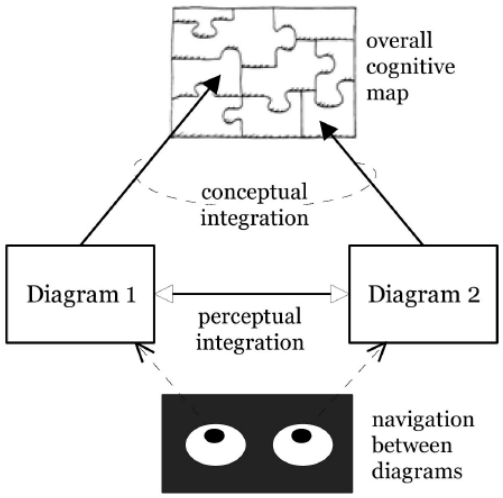


Figure 2.9: Cognitive integration, adapted from [Moo09].

- **Information-carrying variables**: variables used to encode information in a notation.
- **Free variables**: variables not formally used.

A notation with zero visual variables is called non-visual, while a notation that uses all (eight) visual variables is called visually saturated [Moo09]. Figure 2.10 illustrates the visual expressiveness scale.

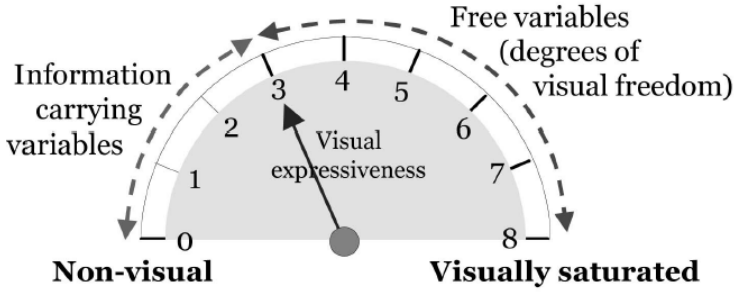


Figure 2.10: Visual expressiveness, adapted from [Moo09].

This principle encompasses the following concepts [Moo09]:

- **Use of colour** - colour is one of the most cognitively effective visual variables, as the human visual system is extremely sensitive to colour variations and can distinguish them quickly and accurately [Mac86; Win93]. Colour differences are detected three times faster than shape and are simpler to remember [Loh93; Tre82].
- **Choice of visual variables** - the choice of visual variables should not be arbitrary but based on the nature of the information to be transmitted [Ber83]. Different visual variables have properties that make them suitable for encoding different types of information.

- **Textual versus graphical encoding** - to maximise visual expressiveness, graphical encoding should be chosen over textual encoding.

2.4.2.7 Dual Coding

Use text to complement graphics, as it is more effective to use text and graphics together to convey information than to use both separately. When information is presented both verbally and visually, representations of that information are encoded into separate systems in working memory and referential connections between the two are strengthened. Also, textual encoding is most effective when used to supplement graphics rather than to replace them [Moo09].

This principle includes the following concepts [Moo09]:

- **Annotations** - including textual explanations (annotations) can improve understanding of diagrams.
- **Hybrid (graphics and text) symbols** - the meaning of graphical symbols can be reinforced and expanded via textual encoding.

2.4.2.8 Graphic Economy

The number of different graphical symbols should be cognitively manageable. This principle concerns graphic complexity, which refers to the number of graphical symbols in a notation, i.e., the size of its visual vocabulary [NC99]. Thus, high graphical complexity significantly reduces the understanding of diagrams, particularly for novice users, since the human ability to discriminate between perceptually distinct alternatives is around six categories [Mil56]. This value sets the upper limit for graphic complexity.

There are three main approaches to handle excessive graphic complexity [Moo09]:

- **Reduce semantic complexity** - simplifying the semantics of a notation provides a way of reducing graphic complexity.
- **Introduce symbol deficit** - graphic complexity can also be reduced by introducing symbol deficit, which means choosing not to show some constructs graphically.
- **Increase visual expressiveness** - instead of reducing the number of symbols increase the human discrimination ability, limited to six symbols.

2.4.2.9 Cognitive Fit

Use different visual dialects for different tasks and audiences. This principle emphasizes the importance of tailoring different representations of information for specific tasks and users [Moo09].

According to this theory, there are at least two reasons for creating multiple visual dialects [Moo09]:

- **Expert-novice differences** - novices have more difficulty discriminating between symbols, lack "chunking" strategies and need to remember the meaning of symbols [BD00; BJ99; KA90; Win93].
- **Representational medium** - the requirements for sketching on whiteboards or paper are different to those for using computer-based drawing tools. To solve this should be used two visual dialects: a simplified visual dialect for sketching and an enriched notation for final diagrams.

2.5 Quality Factors of UML Diagrams

According to Störrle, a diagram element is any line, shape, or textual label that appears in a diagram and can either [Stö16]:

- be positioned within the diagram by itself;
- be shown or hidden by itself;
- contain other diagram elements.

In addition, the following should be considered when counting diagram elements [Stö16]:

- Names can be neither hidden nor moved so they do not count as separate elements. Stereotypes, on the other hand, can be hidden so they do count as labels.
- Adornments with a fixed position relative to the adorned element are not counted, such as arrow heads and aggregation diamonds. Adornments that can be moved include multiplicities, association names, and transition guards.
- Class compartments can be hidden individually, so they count as extra elements unless they are empty.
- Nested elements are counted because they can be hidden and ordered in most tools.

Therefore, the size of a diagram is the number of its diagram elements [Stö16]. Moreover, Störrle found that diagrams containing approximately 20 to 60 diagram elements are optimal for model understanding [Stö14].

Based on the definition of diagram size, Störrle also introduced the concept of diagram flaws and the quality of a diagram. A diagram flaw is defined as any of the following [Stö16]:

1. Bends in lines;
2. Intersections that are visible and not a syntactic element of the language;
3. Touching elements that do not have a close syntactic or semantic association;

4. Sets of merged or aligned lines that are close together unless they have the same type and share exactly one endpoint;

5. Two flaws are fused into one flaw if they are very close together and caused by the same intersecting elements.

The quality of a diagram is the number of flaws it contains [Stö16].

3

Related work

This chapter outlines the three main areas relevant to this research: Model Mining, PoN and Quality Factors of UML Diagrams. A brief overview of each area is provided, highlighting crucial differences and contributions of the present study.

3.1 Model Mining

Hebig et al. [Heb+16] propose a comprehensive methodology for mining software repositories to collect UML models. The goal was to answer when models are created and updated throughout the project's lifespan. This methodology is illustrated in Figure 3.1 and includes the following steps:

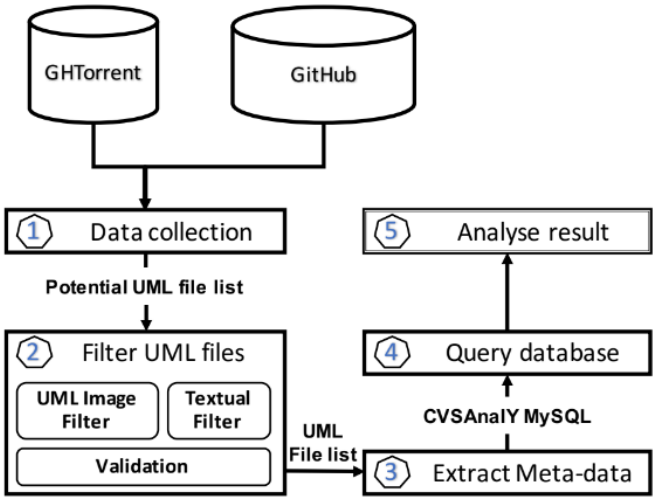


Figure 3.1: Mining process, adapted from [Heb+16].

1. **Data collection** - A list of GitHub repositories is obtained using GHTorrent, excluding those that have been forked or deleted;

2. **Filter UML files** - A semi-automatic process is employed to identify and filter UML files from the collected data. This process includes checking for file extensions

and utilizing machine-learning techniques to classify the files as UML models. Additionally, a manual validation step is implemented to ensure the accuracy of the UML file identification by identifying false positives and negatives;

3. **Extract metadata** - For all projects that contain a UML file, development metadata from the repositories is collected by using the `CVSAnalY` tool [Rob+09];

4. **Query database** - A SQL database is created containing the UML-identified files and their corresponding metadata, allowing for easy and efficient querying;

5. **Analyse result** - The extracted UML models and their corresponding metadata are analyzed to determine patterns in the creation and modification of the models throughout the project's lifespan.

Robles et al. [Rob+17] presents the Lindholmen dataset created using the methodology outlined in [Heb+16]. The dataset comprises a sizable collection of 93,596 UML models, along with the associated metadata of the software repositories to which the UML files belong. The authors made this dataset publicly available to enable researchers to do the following:

- Conducting empirical studies on the benefits and limitations of using UML and modelling in software development;
- Investigating the utilization of UML;
- Developing guidelines for novice UML users;
- Evaluating the effectiveness of various scientific approaches and modelling tools, addressing the limitation of insufficient real cases of models to evaluate newly developed approaches and techniques.

Another study of relevance is [HSA20], in which the authors present a methodology for extracting Business Process Model and Notation (BPMN) models from GitHub software repositories. The authors aimed to create a collection of BPMN models to validate analysis tools for BPMN process modelling. The methodology employed in this study is based on the methods outlined in [Heb+16] and [Rob+17]. Specifically, the authors used a similar mining methodology to that used in these previous studies to extract BPMN models from software repositories.

Savary-Leblanc et al. [Sav+22] present a thorough and quantitative study of UML class diagramming practices based on the analysis of syntactic data from a sample of more than 100,000 UML class diagrams. The goal was to uncover general trends and common practices associated with this type of diagram, including the type of elements used, their frequency, naming conventions, positioning, and possible colouring.

The dataset used in this study was obtained from the GenMyModel [Axe] public repository, known for its adherence to notation standards. To collect the data, the authors

developed a Java application that scraped the public repository of GenMyModel. This application browsed every page of the repository to identify UML diagrams and download the corresponding XMI files. With the files collected, the authors created a second Java application that parsed the resulting XMI files to extract relevant data for analysis. The data extracted included [Sav+22]:

- dimensions of the diagram (width and height);
- true dimensions of the diagram (recomputed from the size and location of elements instead of canvas size);
- information on each element (name, type, width, height, parent element, visibility, case style of the name, colour and diagram identifier);
- coordinates of the center of class owning the maximum elements;
- coordinates of the center of the class most connected to other classes.

The analysis of the data revealed that a classic UML class diagram (from the collected diagrams) has the following characteristics [Sav+22]:

- Between 1 and 20 entities;
- Between 1 and 18 links;
- A linear relationship between the number of links and the number of entities;
- Links are mainly associations, generalizations or compositions;
- Classes have on average between 2 and 3 attributes and between 1 and 2 operations;
- Entities are mostly located on the edges of a rectangle;
- The most important class (in terms of the number of properties or links) is often located at the top left of the diagram;
- Only one colour is used.

The study provides valuable insights into UML class diagramming practices, a methodology for collecting UML class diagrams from a public repository, as well as a method for extracting information from these diagrams.

This dissertation intends to use a web scraping process similar to the one described in [Sav+22] to build a dataset of UML class diagrams. This dataset will be made publicly available since there is a scarcity of datasets that contain this type of diagram.

Furthermore, it seeks to use an automated evaluation tool to identify defects present in the collected UML class diagrams. Finally, analyse the evaluation results to find the most common defects present in the collected dataset.

3.2 Physics of Notations

In [Moo09], Moody proposes a framework of how visual notations communicate and, based on that, develops a set of nine principles to evaluate, compare and design cognitively effective visual notations. These principles were derived from theory and empirical evidence from various fields, mainly outside the Software Engineering field, including the graphic design and communication theory fields. The author performs an in-depth explanation of each principle, highlighting relevant studies and examples of notations that meet or violate them.

Moody and Hillegersberg [MH09] applied the PoN to evaluate the visual syntax of the 13 types of diagrams defined in UML 2.0, including class diagrams. They chose the principles of Semiotic Clarity, Perceptual Discriminability, Semantic Transparency, Visual Expressiveness and Graphic Economy to conduct the evaluation. For each principle, the authors identified violations of the principles and provided recommendations for improvement.

El Kouhen et al. [El +15] analysed the semantic transparency of a subset of UML symbols and compared these standard symbols with alternative solutions where novice users were involved in the design. The results revealed that the newly designed symbols showed better semantic transparency when compared to the standard notation.

In this dissertation, the principles of PoN proposed in [Moo09] provide a valuable framework for evaluating UML class diagrams. However, it should be noted that only a subset of these principles, namely Perceptual Discriminability, Visual Expressiveness, Dual Coding, and Graphic Economy, will be applied in this research.

The objective of this study is not to evaluate the visual notation of UML class diagrams, as done in previous studies such as [MH09] and [El +15]. Instead, this study employs the principles of PoN to evaluate a dataset of UML class diagrams to identify defects present in these diagrams.

3.3 Quality Factors of UML Diagrams

Störrle examines the relationship between the size of UML diagrams and modeller performance [Stö14]. He hypothesises that there is a negative correlation between diagram size and modeller performance. He also hypothesises that layout quality becomes more important as diagram size increases, since larger diagrams require a better layout to be easily understood.

To test this hypothesis, Störrle analysed the data in a study involving 78 participants and over 1200 measurements. Several performance indicators were taken into consideration, such as errors, preference/assessment, and cognitive load. The results of the study support the hypothesis, revealing a negative correlation between diagram size and modeller performance. Furthermore, the study suggests that layout quality becomes increasingly important for larger diagrams.

Based on the study's findings, Störrle concludes that diagrams containing approximately 20 to 60 diagram elements are optimal for model understanding.

In [Stö16], Störrle built upon the previous work presented in [Stö14] to address limitations in terms of the validity and generalizability of the findings and to refine the definition of diagram quality. He established three objectives to achieve this:

- To increase validity by analyzing a larger and more diverse dataset of 60 diagrams of the five most widely used UML diagram types, with 156 participants.
- To broaden generalizability by adding two more UML diagram types.
- To further analyze diagram quality.

To meet these objectives, the author improved the definition of diagram size and introduced the definition of topographic layout quality. A diagram element is defined as any line, shape, or textual label present in a diagram that can either [Stö16]:

- be positioned within the diagram independently;
- be shown or hidden independently;
- contain other diagram elements.

Therefore, the size of a diagram is defined as the number of its diagram elements. When counting diagram elements, the following aspects should be considered [Stö16]:

- Names can be neither hidden nor moved so they do not count as separate elements. Stereotypes, on the other hand, can be hidden so they do count as labels.
- Adornments with a fixed position relative to the adorned element are not counted, such as arrow heads and aggregation diamonds. Adornments that can be moved include multiplicities, association names, and transition guards.
- Class compartments can be hidden individually, so they count as extra elements unless they are empty.
- Nested elements are counted because they can be hidden and ordered in most tools.

Based on the definition of diagram size, the author introduced the concept of diagram quality. A diagram flaw is defined as any of the following [Stö16]:

1. Bends in lines;
2. Intersections that are visible and not a syntactic element of the language;
3. Touching elements that do not have a close syntactic or semantic association;

4. Sets of merged or aligned lines that are close together unless they have the same type and share exactly one endpoint;

5. Two flaws are fused into one flaw if they are very close together and caused by the same intersecting elements.

Finally, the topological quality of a diagram is defined as the number of flaws it contains.

The results of [Stö16] confirmed the findings of [Stö14] and provided guidelines for creating better diagrams. It was suggested that the flaw rate of a diagram should not exceed 0.5 and the number of flaws should not exceed 15-20. Additionally, the author recommended prioritizing reducing the number of flaws over reducing the diagram size.

This dissertation aims to identify defects present in collected UML class diagrams using the principles of diagram size and diagram flaws presented in [Stö14] and [Stö16].

4

Data Collection

This chapter presents the data collection process used to create the UML class diagram dataset analyzed in this dissertation. It begins by highlighting the challenges encountered. It then shows where the models were collected from. Additionally, it shows the web scraper solution, the relevant technologies used to implement the web scraper, and the details of its implementation. Lastly, it presents the results obtained by using the web scraper. The complete source code of the web scraper is available in the following GitHub repository link.

4.1 Challenges

The main challenge encountered during the data collection process was related to the format of the models that would be acceptable for conducting this study. Specifically, extracting information from UML class diagrams presents a challenge for researchers who aim to analyze a dataset of these diagrams. This is because they are usually represented in image formats, such as .png, .jpeg, or .svg, making data extraction difficult.

To address this issue, a tool called Img2UML [KC13] was developed to extract UML models from images and produce an XMI file of the UML model. This tool employs an image recognition technique to identify and organize the various diagram elements into an XMI file. However, it has limitations, such as not being able to extract all details from images and its performance being affected by the resolution of the images.

To overcome these limitations, this dissertation focuses solely on analyzing UML class diagrams with the XMI format, as previously done in [Sav+22]. This format provides a structured representation of the UML class diagram, allowing for the easy extraction of all model information through parsing its tags.

Therefore, this approach offers a better method for extracting information from UML class diagrams when compared to extracting information from image formats.

The second challenge faced was running the web scraper program locally, as the program would require a lot of heavy computational operations. As a solution, we decided to use the Google Colab platform for running the web scraper. More details about

this platform and why we chose it are described in subsection 4.4.2.

4.2 Data Source

The main objective of this dissertation is to investigate common defects in UML class diagram modelling. To achieve this goal, a dataset of UML class diagrams is required. The chosen approach was to create this dataset using web scraping to collect these models.

To select an appropriate data source, the discussion in section 4.1 was taken into consideration. Thus, the models needed to be downloadable in the XMI format. Also, they had to adhere to the notation standard. Additionally, it was crucial to collect a substantial number of models to ensure sufficient data for analysis.

Consequently, the GenMyModel platform was chosen to extract the UML class diagrams. GenMyModel is a web-based modelling platform that supports various notations, including UML, BPMN, and Archimate. Also, unlike other online platforms, it adheres to notation standards. Moreover, it hosts a publicly accessible repository of its user's projects, which currently has 1,566,214 public models as of September 2023 [Axe].

The platform's repository allows searching for project names and notations, as shown in Figure 4.1. However, one downside is that it does not allow searching for specific types of UML diagrams, such as UML class diagrams. This verification needed to be conducted later during the web scraping process.

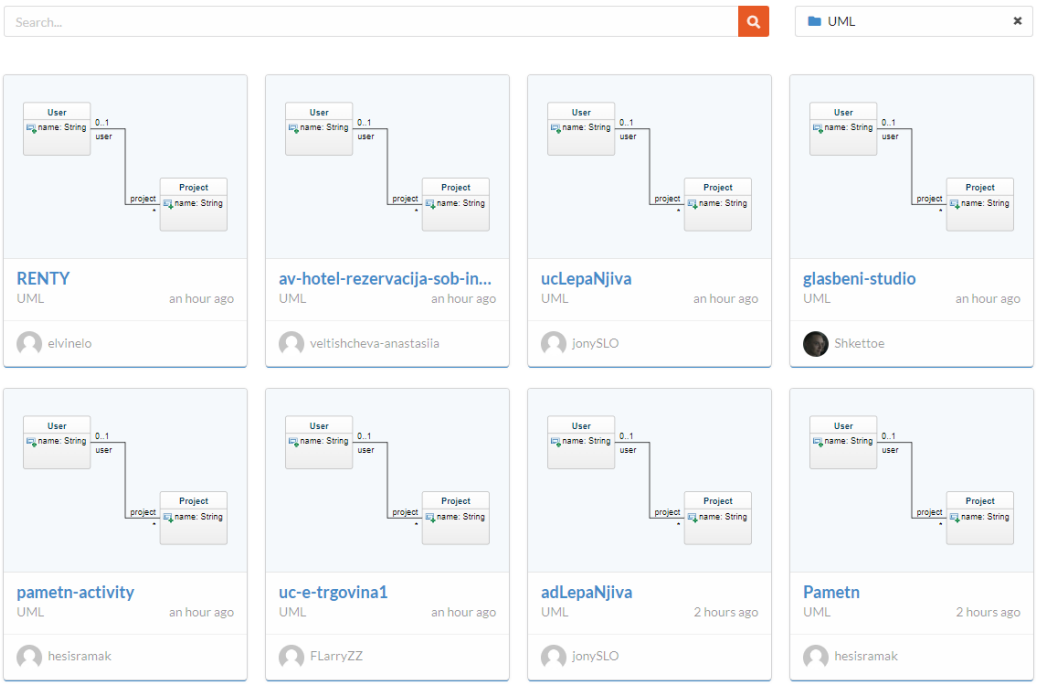


Figure 4.1: GenMyModel public repository.

4.3 Web Scraper Solution

The web scraper execution process is illustrated in Figure 4.2 and involves the following steps:

1. **Fetch a specific page from the GenMyModel public repository** - send an HTTP `GET` request to obtain the HTML document for a specific page of the repository.
2. **Find the URLs of the projects pages** - parse the HTML document of the previous step to identify the cards of the project's pages and extract their URLs.
3. **Fetch the page of each project** - retrieve the HTML document of each project page by sending an HTTP `GET` request to each one.
4. **Find the ID of each project** - parse the HTML document of each project page to extract the project ID.
5. **Fetch the XMI file of each project** - send an HTTP `GET` request to get an HTML document containing the contents of the XMI file for each project.
6. **Check if the XMI files contain a UML class diagram** - parse the XMI tags in each file to verify if it contains UML class diagram elements.
7. **Download the UML class diagrams** - save the XMI files.

4.4 Relevant Technologies

To create and run the web scraper, we opted for the Python programming language and used Google Colab as the platform to run the program. This topic explains what these technologies are and why we chose them.

4.4.1 Python

In order to create the web scraping program to scrape XMI files containing a UML class diagram from GenMyModel, it was necessary to select a programming language to be used for the development of the program.

The programming language chosen to create this web scraper was Python. Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. This language is known for its simplicity and ease of use, and it also supports modules and packages that promote code reuse and program modularity [Pyt].

Specifically, the `requests` library [Req] for making HTTP requests and the `BeautifulSoup` library [Soua] for parsing HTML documents are useful Python libraries when creating a web scraping program and were crucial in choosing this programming language.

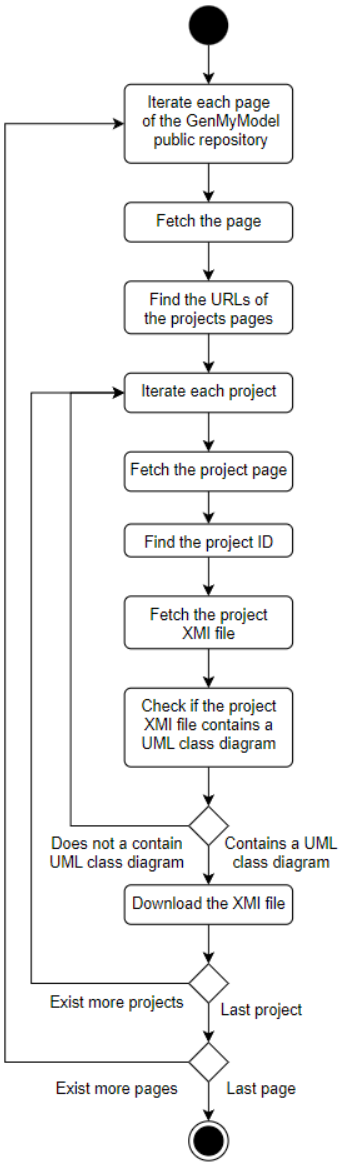


Figure 4.2: Activity diagram of the web scraper execution process.

4.4.2 Google Colab

Google Colab [Resb], short for Google Colaboratory, is a cloud-based platform provided by Google. This platform allows users to write and execute Python code through the web browser, eliminating the need for local machine setup or resource-intensive hardware. It is particularly beneficial for those working in machine learning, data analysis, and education. Essentially, Google Colab is a hosted Jupyter Notebook [Jup] service that does not require any setup and provides free access to computing resources, including GPUs [Resa].

Moreover, Google Colab makes it easy to create, share, and collaborate on Python-based projects that require heavy computational resources. It uses Google's cloud infrastructure to provide access to powerful GPUs and TPUs, significantly accelerating tasks involving compute-intensive operations [Resa].

Some of the key advantages of using this platform include [Gee]:

- **Free access to computing resources** - Free Colab users get access to GPU and TPU runtimes for up to 12 hours without any charges.
- **Notebook sharing** - Colab notebooks can be easily shared with others, allowing collaborators to view and run the code without the need for extra software or environment replication. Notebook sharing can be done through links or via Google email addresses.
- **Pre-installed libraries** - Google Colab offers multiple pre-installed libraries, reducing the setup time. Such libraries include NumPy [Num], Pandas [Pan], Matplotlib [Mat], PyTorch [Fou], TensorFlow [Ten], Keras [Ker], and more.
- **Collaborative coding** - Multiple collaborators can simultaneously edit a notebook, updating it automatically as the team codes.
- **Google Drive integration** - Google Colab allows users to store their notebooks and datasets directly in their Drive accounts. This integration makes it easier to work on projects across different devices. Additionally, it serves as a backup of the data from any disasters.
- **GitHub integration** - Google Colab allows users to link their GitHub account to import and export code files.
- **Automatic version control** - Google Colab tracks all the changes made since the file creation.

Although Google Colab has several benefits, it also has some limitations [Resa]:

- **Session lifetime** - Colab sessions have a limited lifetime. Long-running processes will be interrupted if the session duration exceeds the platform's time limit of 12 hours.
- **Resource availability** - Resource availability, such as GPUs and TPUs, is not guaranteed and may vary depending on demand.

Running the web scraper program locally presented a challenge as the program would make a large number of HTTP requests and also would be necessary to store a large number of files as a result of the web scraping process. Due to this fact, we decided to use the cloud to execute this program by resorting to the Google Colab platform. We opted for this platform due to the advantages previously mentioned. Concretely, it allows the execution of Python programs, and there is no need to import libraries since this platform already provides numerous pre-installed libraries. Additionally, it offers free access to powerful computational resources for up to 12 hours of runtime. Finally, it has Google Drive integration, allowing to store the scraped files directly to this service, thus making it easier to share the collected dataset with others.

4.5 Implementation

A Python application was developed to extract UML class diagrams from public GenMyModel projects. This application consists of two Java files: `uml_class_diagram_scraper.py` and `utils.py`. The first file contains the application's main function and is the executable file (Listing 4.1). The latter contains auxiliary functions that are called by the main function.

It is necessary to initialize the script with the starting and ending page numbers of the GenMyModel repository (`INITIAL_PAGE` and `FINAL_PAGE`) to establish a range of pages that the web scraper will process. By doing so, the application can target and retrieve the UML class diagrams of the projects from the specified range of pages. To run this tool, the following command format must be used: `python uml_class_diagram_scraper.py INITIAL_PAGE FINAL_PAGE`. For example, to process pages 1 to 300 of the GenMyModel repository, the corresponding command is as follows: `python uml_class_diagram_scraper.py 1 300`.

Listing 4.1: Web scraper main function code.

```
def main():

    # Ensure the correct use
    if len(sys.argv) != 3:
        sys.exit("Usage: python uml_class_diagram_scraper.py INITIAL_PAGE FINAL_PAGE")

    INITIAL_PAGE = int(sys.argv[1])
    FINAL_PAGE = int(sys.argv[2])

    # Check if the end page number is smaller than the start page
    if FINAL_PAGE < INITIAL_PAGE:
        sys.exit("The final page must be greater or equal than the initial page!")

    # Page range
    pages = range(INITIAL_PAGE, FINAL_PAGE + 1)

    project_ids = []

    # For each public repository page, get the projects
    for page in pages:
        projects = utils.get_projects_of_page(page)

        # Get the ID of each project
        for project in projects:
            project_id = utils.project_id(project)

            # Check if the project ID is valid
            if project_id is not None:
                project_ids.append(project_id)

    # Download the XMI for each project
```

```
    for project_id in project_ids:
        is_uml_class_diagram, content = utils.is_UML_class_diagram(project_id)

        # Verify that the XMI file is a UML class diagram
        if is_uml_class_diagram:

            # Check if the project has already been downloaded
            if not utils.is_project_downloaded(project_id):

                # Download the project's XMI file
                utils.download_xmi(content, project_id)

                # Save the downloaded project ID
                utils.save_downloaded_project_id(project_id)
```

Once the page range is defined, the program proceeds to iterate over each page, searching for the projects within that page. To accomplish this, the `get_projects_of_page` function (Listing 4.2) is called. Initially, an HTTP `GET` request is made to the targeted page using the `requests` library, and its contents are retrieved. Subsequently, the `BeautifulSoup` library is used to parse the HTML document and find all the project cards present on that page. These project cards are identified by locating tags denoted by <a> with the class "`ui card public-project blue`", as shown in Figure 4.3. Lastly, for each card, the value of the "`href`" attribute is extracted to obtain the URL of each project.

Listing 4.2: `get_projects_of_page` function code.

```
def get_projects_of_page(page):

    # Create the page URL
    url = change_page(page)

    # Make the HTTP GET request to the URL
    response = requests.get(url)

    # Check if the HTTP GET request was not successfully answered
    if response.status_code != 200:
        return None

    # Get the page content
    content = response.content

    # Parse the page content
    soup = BeautifulSoup(content, 'html.parser')

    # Find all "a" tags with the "ui card public-project blue" class
    links = soup.find_all('a', class_='ui card public-project blue')

    # Extract the project's URLs
    project_urls = [URL + link['href'] for link in links]

```

```
25      return project_urls
```

```
<a class="ui card public-project blue" href="/api/repository/nelsoncarabeo9/Caso_de_uso">...</a>
<a class="ui card public-project blue" href="/api/repository/luisafdm0910/Diagrama_de_secuencias">...</a>
<a class="ui card public-project blue" href="/api/repository/luisafdm0910/Diagrama_de_secuencia_-_consultar">...</a>
<a class="ui card public-project blue" href="/api/repository/gestionpla2/Gestión_de_plan_de_estudios">...</a>
<a class="ui card public-project blue" href="/api/repository/gilsiellengomes/Locar-Fitas.ModelagemDeSoftware">...</a>
<a class="ui card public-project blue" href="/api/repository/76235294/Empresa_Horizonte">...</a>
<a class="ui card public-project blue" href="/api/repository/bdmenar/Diagrama_de_Clases_Hotel">...</a>
<a class="ui card public-project blue" href="/api/repository/bdmenar/Banco_clases">...</a>
<a class="ui card public-project blue" href="/api/repository/zichenghuang19/ITP">...</a>
<a class="ui card public-project blue" href="/api/repository/joseantonio.drm/AutoridadPortuaria">...</a>
<a class="ui card public-project blue" href="/api/repository/jj1011/Diagrama_de_casos_de_uso1">...</a>
<a class="ui card public-project blue" href="/api/repository/jelena.mihalek123/DijagramKlasaPIS1">...</a>
<a class="ui card public-project blue" href="/api/repository/CRexOnline/DSASP">...</a>
<a class="ui card public-project blue" href="/api/repository/ognjenjov02/hotelClassDiagram">...</a>
<a class="ui card public-project blue" href="/api/repository/dhawalnil/Job_Portal_System">...</a>
```

Figure 4.3: Project cards HTML code example.

Upon obtaining the URLs for each project, the program proceeds to acquire their respective project IDs. To accomplish this task, the `project_id` function (Listing 4.3) is called for each project. This function takes as input the project's URL and makes an HTTP GET request to the corresponding page to retrieve its contents. The `BeautifulSoup` library is subsequently used to parse the HTML document and find the project ID. The project ID is identified by searching for the <meta> tag with the "`property`" attribute set to "`og:image`". Finally, a regular expression is applied to extract the ID from the "`content`" attribute of the previously identified tag, which can be found after the string "https://app.genmymodel.com/api/projects/", as depicted in Figure 4.4.

Listing 4.3: `project_id` function code.

```
1  def project_id(project_url):
2
3      # Make the HTTP GET request to the URL
4      response = requests.get(project_url)
5
6      # Check if the HTTP GET request was not successfully answered
7      if response.status_code != 200:
8          return None
9
10     # Get the page content
11     content = response.content
12
13     # Parse the page content
14     soup = BeautifulSoup(content, 'html.parser')
15
16     # Get the "meta" tag with "property" attribute equal to "og:image"
17     og_image_tag = soup.find('meta', {'property': 'og:image'})
18
19     # Extract the project ID using a regular expression
20     try:
21         project_id = re.search(r'/projects/([^/]+)/', og_image_tag['content'])
```

```
    except:
        return None

    return project_id.group(1)
```

```
<meta property='og:image' content='https://app.genmymodel.com/api/projects/_4h2CQNu9Ee2JhIahQQsnAA/diagrams/_4h5FkNu9Ee2JhIahQQsnAA/jpeg'/>
```

Figure 4.4: Meta tag HTML code example.

Next, each project's XMI file is verified to see if it contains a UML class diagram. Note that each project has an XMI file that can contain elements from different types of UML diagrams. This is because the GenMyModel editor allows mixed use of class, component, deployment, object and use case diagram entities, as shown in Figure 4.5.

For this, the `is_UML_class_diagram` function (Listing 4.4) is called. This function takes the project ID as input and uses it to create the URL where the XMI file of that project is located. After generating the URL, an HTTP `GET` request is made to fetch the XMI file. The `BeautifulSoup` library is then used to parse the XMI file and to find essential elements in a UML class diagram, i.e., class, interface, datatype and enumeration. Finally, the function checks whether any of these elements were found in the XMI file. If any of these elements are present, it indicates that the XMI file contains a UML class diagram and returns `True`, else returns `False`.

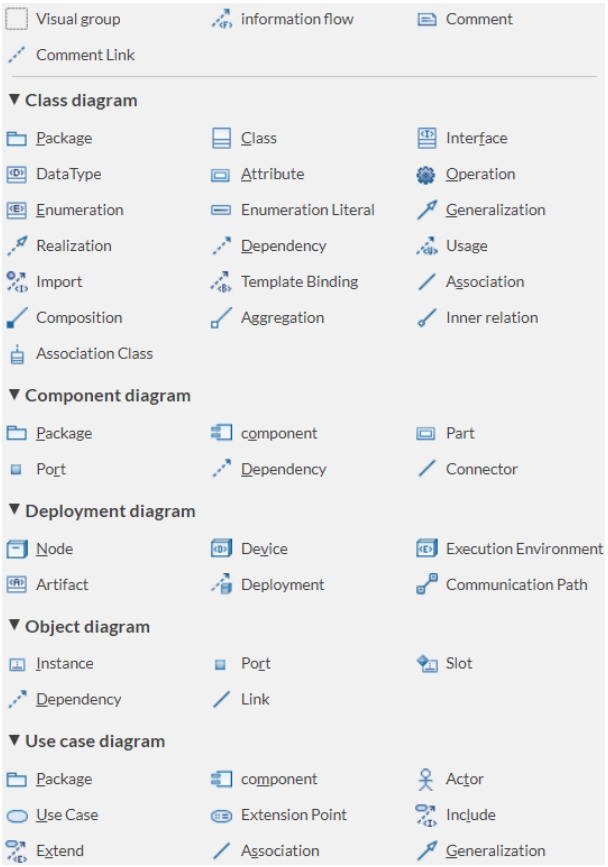


Figure 4.5: GenMyModel editor for UML class diagrams.

Listing 4.4: `is_UML_class_diagram` function code.

```
UML_CLASS_DIAGRAM_ELEMENTS = ["uml:Class", "uml:Interface", "uml:DataType", "uml:
    ↪ Enumeration"]

def is_UML_class_diagram(project_id):

    # Create the project's XMI URL
    url = project_xmi(project_id)

    # Make the HTTP GET request to the URL
    response = requests.get(url)

     # Check if the HTTP GET request was not successfully answered
    if response.status_code != 200:
        return False, None

    # Get the page content
    content = response.content

    # Parse the page content
    soup = BeautifulSoup(content, 'xml')

    # Check if the XMI file contains the essential elements of a UML class diagram
    elements = soup.find_all('packagedElement', {'xsi:type': UML_CLASS_DIAGRAM_ELEMENTS})

    if elements:
        return True, content
    else:
        return False, None
```

Subsequently, for each project, if the XMI file contains a UML class diagram and it has not yet been downloaded, it will be downloaded using the `download_xmi` function (Listing 4.5).

Listing 4.5: `download_xmi` function code.

```
def download_xmi(content, project_id):

    filename = XMI_PATH + project_id + ".xmi"

    # Save the XMI file
    with open(filename, "wb") as file:
        file.write(content)

    return filename
```

As a last step, the project ID of each downloaded XMI file is saved into a text file called `downloaded_files.txt` to avoid downloading files already present in the dataset. This is done using the `save_downloaded_project_id` function (Listing 4.6).

Listing 4.6: `save_downloaded_project_id` function code.

```
def save_downloaded_project_id(project_id):

    # Write project IDs to the text file
    with open("downloaded_files.txt", "a") as file:
            file.write(project_id + "\n")
```

4.6 Results

The web scraper program (described in section 4.5) was executed on the Google Colab platform. It is important to note that the downloaded XMI files were stored in a Google Drive account, as Google Colab allows the integration of this service. An example of the downloaded XMI files is presented in Listing 4.7.

Listing 4.7: Example of downloaded XMI files.

```
<?xml version="1.0" encoding="UTF-8"?>
  <uml:Model xmi:id="f0e4e3a0-553f-44f9-80c6-332e5e0e68ff" name="model">
  (...)
    <packagedElement xsi:type="uml:Package" xmi:id="_PPuRoP7OEDGQjpANbhXayQ" name="
        ↪ ShoppingCartExample">
      <xmi:Extension extender="http://www.eclipse.org/emf/2002/Ecore">
        <eAnnotations xmi:id="_PPuRoP7OEDGQjpANbhXayQ0" source="genmymodel">
          <details xmi:id="_PPuRoP7OEDGQjpANbhXayQ00" key="uuid" value="
              ↪ _PPuRoP7OEDGQjpANbhXayQ"/>
        </eAnnotations>
      </xmi:Extension>
      <packagedElement xsi:type="uml:Class" xmi:id="_RPRHgP7OEDGQjpANbhXayQ" name="
          ↪ ShoppingCart">
        <xmi:Extension extender="http://www.eclipse.org/emf/2002/Ecore">
          <eAnnotations xmi:id="_RPRHgP7OEDGQjpANbhXayQ0" source="genmymodel">
            <details xmi:id="_RPRHgP7OEDGQjpANbhXayQ00" key="uuid" value="
                ↪ _RPRHgP7OEDGQjpANbhXayQ"/>
          </eAnnotations>
        </xmi:Extension>
        <ownedAttribute xmi:id="_v6q9gP7REDGNKryMNR39Ag" name="creationDate">
          <xmi:Extension extender="http://www.eclipse.org/emf/2002/Ecore">
            <eAnnotations xmi:id="_v6q9gP7REDGNKryMNR39Ag0" source="genmymodel">
              <details xmi:id="_v6q9gP7REDGNKryMNR39Ag00" key="uuid" value="
                  ↪ _v6q9gP7REDGNKryMNR39Ag"/>
            </eAnnotations>
          </xmi:Extension>
          <type xsi:type="uml:PrimitiveType" href="pathmap://GENMYMODEL_LIBRARIES/
              ↪ GenMyModelPrimitiveTypes.library.uml#//Date"/>
        </ownedAttribute>
      </packagedElement>
      <packagedElement xsi:type="uml:Class" xmi:id="_SWcyAP7OEDGQjpANbhXayQ" name="Order">
        <xmi:Extension extender="http://www.eclipse.org/emf/2002/Ecore">
```

```
27          <eAnnotations xmi:id="_SWcyAP7OEDGQjpANbhXayQ0" source="genmymodel">
28            <details xmi:id="_SWcyAP7OEDGQjpANbhXayQ00" key="uuid" value="
                  ↪ _SWcyAP7OEDGQjpANbhXayQ"/>
29          </eAnnotations>
30        </xmi:Extension>
31        <ownedAttribute xmi:id="_mWHeEP7REDGNKryMNR39Ag" name="id">
32          <xmi:Extension extender="http://www.eclipse.org/emf/2002/Ecore">
33            <eAnnotations xmi:id="_mWHeEP7REDGNKryMNR39Ag0" source="genmymodel">
34              <details xmi:id="_mWHeEP7REDGNKryMNR39Ag00" key="uuid" value="
                    ↪ _mWHeEP7REDGNKryMNR39Ag"/>
35            </eAnnotations>
36          </xmi:Extension>
37          <type xsi:type="uml:PrimitiveType" href="http://www.omg.org/spec/UML/20131001/
                ↪ PrimitiveTypes.xmi#//Integer"/>
38        </ownedAttribute>
39      </packagedElement>
40    (...)
41  </uml:Model>
42 </xmi:XMI>
```

The resulting dataset contains 103,103 files in the XMI format, all containing a UML class diagram. Altogether, the size of this dataset is 19.9 GB, with an average file size of 193 KB. This dataset is available in the following Google Drive link.

5

Principles for the Evaluation of UML Class Diagrams

This chapter presents and describes the principles chosen to evaluate the collected UML class diagrams. Specifically, these principles encompass elements of the PoN [Moo09] and the principles of diagram size and diagram flaws, as described in [Stö16]. Additionally, it illustrates these principles with examples, demonstrating instances where the principles are respected and not respected.

5.1 The Physics of Notations Principles

The collected UML class diagrams will be evaluated for defects based on the principles of the PoN [Moo09], described in section 2.4. Specifically, the principles of Semiotic Clarity, Perceptual Discriminability, Visual Expressiveness, and Graphic Economy were chosen for this evaluation.

5.1.1 Visual Expressiveness

This principle states that the full range and capabilities of the visual variables should be used [Moo09] (subsection 2.4.2.6).

The evaluation will employ the concept of the use of colour. Colour is one of the most cognitively effective of all visual variables: the human visual system is highly sensitive to variations in colour and can quickly and accurately distinguish between them. Notably, colour differences are detected three times faster than shape and are also more easily remembered [Moo09]. Therefore, this evaluation aims to verify whether the graphical elements within the diagrams conform to this principle by employing colours beyond their default colour.

Figure 5.1 shows an example in which this principle is respected since the class uses the colour attribute. In contrast, Figure 5.2 shows an example in which this principle is not respected since the class uses the default colour.

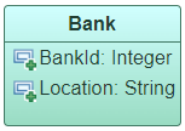


Figure 5.1: Example of the Visual Expressiveness principle respected.

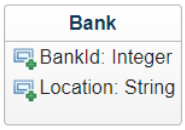


Figure 5.2: Example of the Visual Expressiveness principle not respected.

5.1.2 Dual Coding

This principle dictates that text should be used to complement graphics. Therefore, using text and graphics together to convey information is more effective than using each separately [Moo09] (subsection 2.4.2.7).

The evaluation will use the concept of Hybrid Symbols, in which the hybrid symbols use textual and graphical elements. In the hybrid representation, the text both expands and reinforces the meaning of the graphics [Moo09]. Thus, this evaluation of UML class diagrams checks whether the relationship elements use this hybrid representation, more concretely, whether relationships contain names and multiplicities (if applicable).

Figure 5.3 shows an example in which this principle is respected since the association includes names and a multiplicities. On the other hand, Figure 5.4 shows an example in which this principle is not respected since the association has no names or multiplicities, that is, there is no text to complement the graphics.

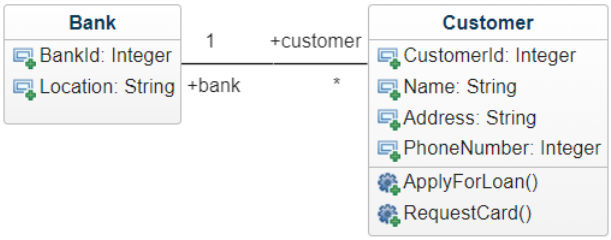


Figure 5.3: Example of the Dual Coding principle respected.

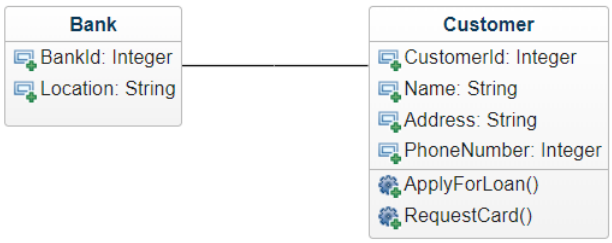


Figure 5.4: Example of the Dual Coding principle not respected.

5.1.3 Graphic Economy

This principle defines that the number of different graphical symbols must be cognitively manageable [Moo09] (subsection 2.4.2.8). The human ability to discriminate between perceptually distinct alternatives revolves around six categories, this being the upper limit for graphical complexity [Mil56; Moo09]. Consequently, in this evaluation, the diagrams will be examined to determine whether they exceed the limit of using more than six different graphical symbols and, if so, by how much.

Figure 5.5 shows an example in which this principle is respected since the diagram has two different graphical symbols (class and association), not exceeding the limit of six. In contrast, Figure 5.6 shows an example in which this principle is not respected since the diagram has seven different graphical symbols (package, class, enumeration, generalization, association, composition and aggregation), exceeding the limit of six.

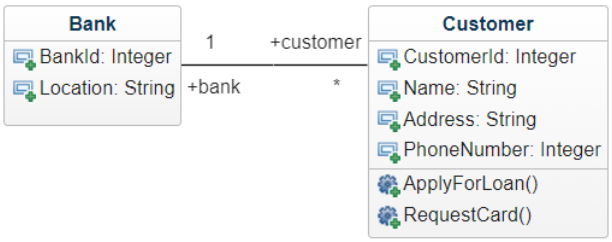


Figure 5.5: Example of the Graphic Economy principle respected.

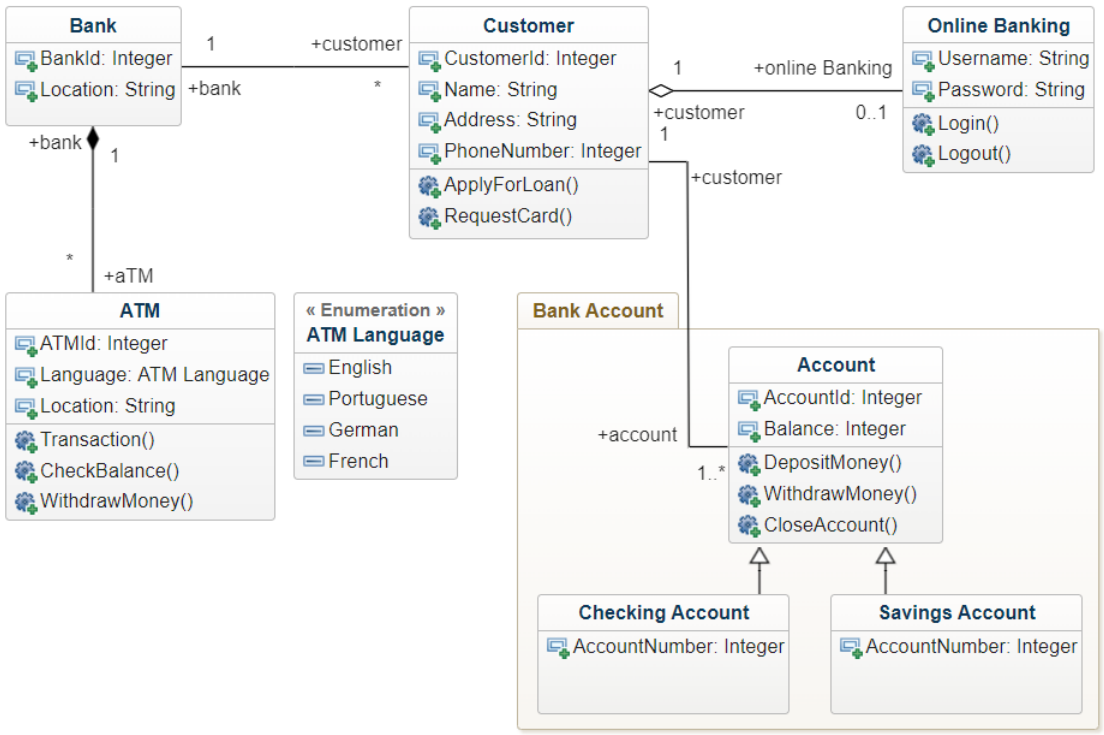


Figure 5.6: Example of the Graphic Economy principle not respected.

5.1.4 Perceptual Discriminability

This principle defines that graphical symbols should be distinguishable from each other [Moo09] (subsection 2.4.2.2).

The evaluation takes into account the concept of visual distance, which is measured by the number of visual variables in which graphical symbols differ [Moo09]. When examining UML class diagrams, it becomes apparent that its elements exhibit a low visual distance, as the symbols differ in three visual variables (shape, brightness, and texture). Nevertheless, within the scope of this evaluation, the visual distance serves as a means to ascertain whether the symbols employed are distinct from one another.

Furthermore, the principle of redundant coding is employed, stating that multiple visual variables must be used to differentiate the symbols. In this case, colour can be used to increase the discriminability among elements [Moo09]. Consequently, this evaluation incorporates an examination of the colour attributes of the elements to verify their distinguishability.

Lastly, the principle of textual differentiation is employed as a means of distinguishing between symbols [Moo09]. Thus, this evaluation checks the symbol's textual elements to ascertain that the symbols are distinguishable. Specifically, names, attributes, operations, and enumeration literals (if applicable) will be checked.

Figure 5.7 shows an example in which this principle is respected since the two classes can be differentiated by colour, as well as by the names, attributes and operations they contain. In contrast, Figure 5.8 shows an example in which this principle is not respected since the two classes cannot be differentiated.

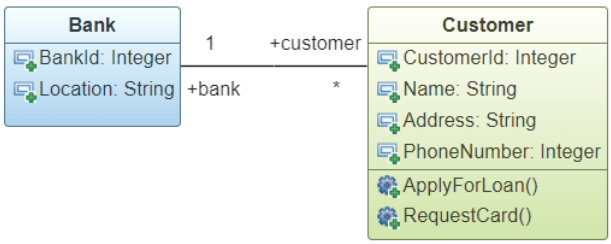


Figure 5.7: Example of the Perceptual Discriminability principle respected.

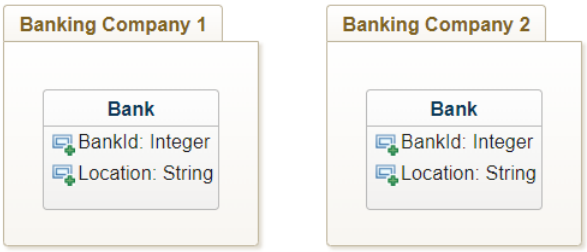


Figure 5.8: Example of the Perceptual Discriminability principle not respected.

5.2 Quality Factors of UML Diagrams Principles

In addition to the principles of PoN, the evaluation conducted in this study also incorporates the principles of diagram size and diagram flaws, as proposed in [Stö16].

The principle of diagram size is defined as the total number of diagram elements, and its ideal range should be between 20 and 60 elements. Note that a diagram element is any line, shape, or textual label present in the diagram that can be independently positioned, shown or hidden, or contains other diagram elements [Stö16]. Consequently, the analysis of the diagrams aims to determine whether they meet this preferred range and, if not, by how much they deviate.

A diagram flaw is defined as any of the following principles [Stö16]:

1. **Close Merged or Aligned Lines principle** - sets of merged or aligned relationship lines should not be close unless they have the same type and share exactly one endpoint. Figure 5.9 shows an example in which this principle is not respected since there is a set of relationship lines that are close together, don't have the same type and don't share one endpoint.

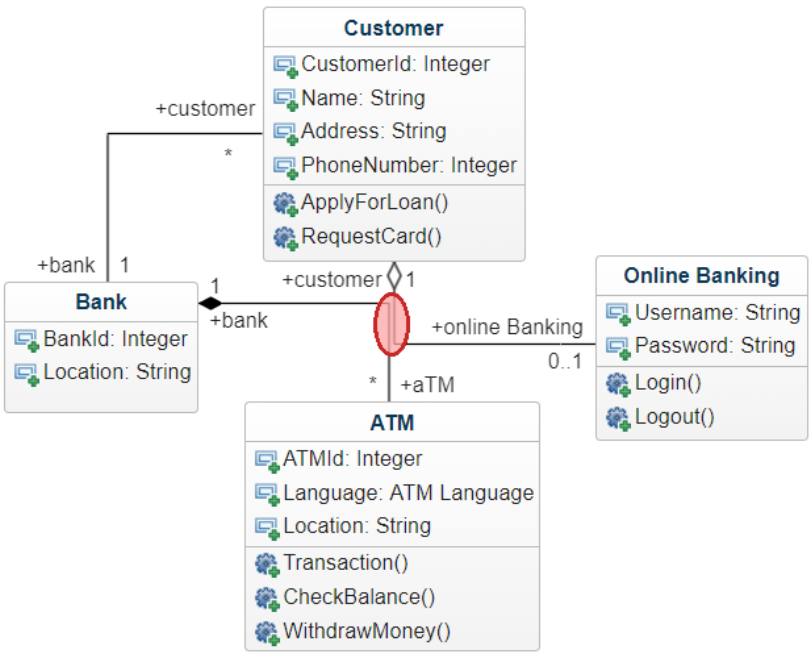


Figure 5.9: Example of the Close Merged or Aligned Lines principle not respected.

2. **Bends in Lines principle** - relationships should not have bends in their lines. Figure 5.10 shows an example in which this principle is respected since there are no bends in the relationship line. On the contrary, Figure 5.11 shows an example in which this principle is not respected since there is a bend in the relationship line.

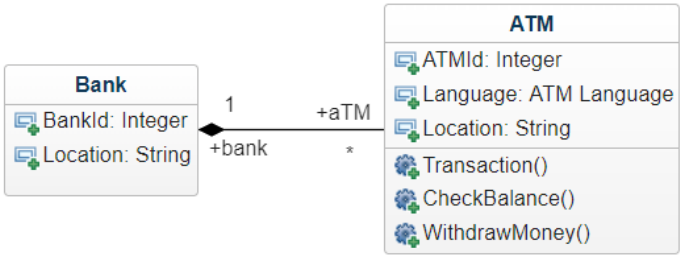


Figure 5.10: Example of the Bends in Lines principle respected.

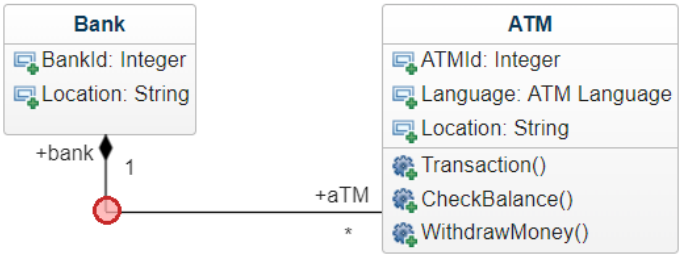


Figure 5.11: Example of the Bends in Lines principle not respected.

3. **Intersections principle** - there should be no intersections between relationships or relationships and elements. Figure 5.12 shows an example in which this principle is not respected since there is an intersection between relationships.

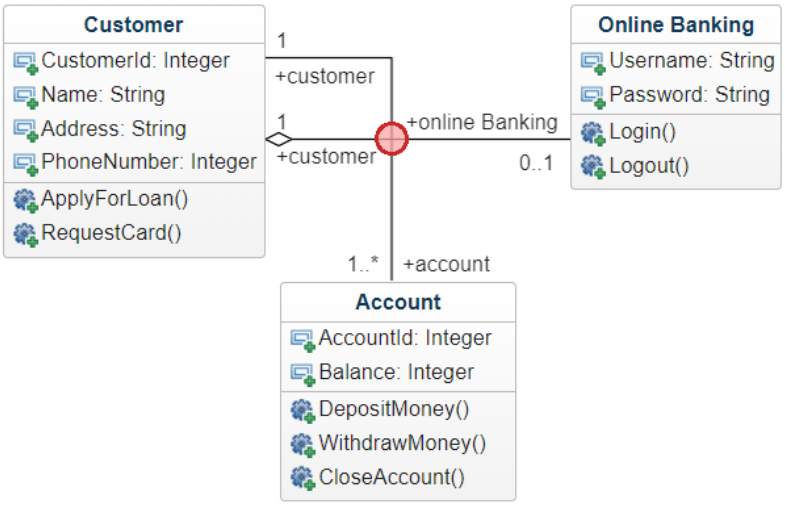


Figure 5.12: Example of the Intersections principle not respected.

4. **Overlapping Elements principle** - elements should not overlap each other. Figure 5.13 shows an example in which this principle is not respected due to overlapping elements.

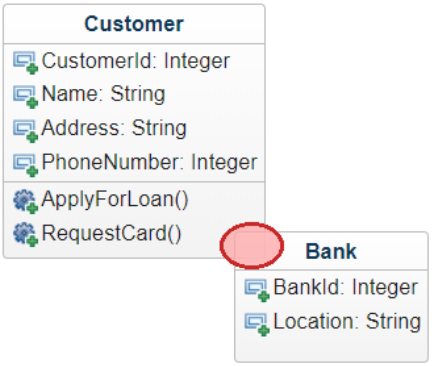


Figure 5.13: Example of the Overlapping Elements principle not respected.

Thus, the evaluation process involves analyzing the diagrams to ascertain whether they exhibit any diagram flaws and, if so, which ones and how many.

6

Evaluation Process

This chapter presents an overview of the evaluation process conducted in this dissertation. Firstly, it presents a description of the evaluation process. Furthermore, it outlines the overall evaluation process.

6.1 Description

As previously mentioned, the main objective of this work is to identify the most typical defects present in the collected UML class diagrams. To achieve this goal, an automated evaluation process is employed to detect and classify these defects within the diagrams.

Initially, a comprehensive dataset was collected, consisting of over a hundred thousand UML class diagrams obtained from the GenMyModel public repository projects (as discussed in chapter 4).

Furthermore, chapter 5 focuses on identifying and describing the principles used to assess these models and identify defects.

This section describes the evaluation process without delving too deeply into the specifics of the implementation process, covered in chapter 7. The subsequent section provides a high-level overview of the overall evaluation process, outlining its principal stages.

Hence, the overall evaluation process consists of running an automated evaluation tool to assess each UML class diagram in the dataset, identifying the defects by checking whether each principle (described in Chapter 5) is respected or violated. Subsequently, after evaluating the entire dataset, it becomes possible to discern the predominant defects that manifest in these diagrams.

6.2 Overall Evaluation Process

The overall evaluation process is depicted in Figure 6.1, illustrating its four main steps.

The initial step involves using a Python application to evaluate each XMI file in the dataset. So, with the principles presented in Chapter 5, each model is evaluated to verify

whether it respects or violates each principle. To accomplish this, the XMI files are parsed to obtain the models information. As an illustration, Figure 6.2 shows a UML class diagram, and Listing 6.1 provides a snippet of the corresponding XMI code. The XMI format provides a structured representation of the UML class diagram, allowing for the easy extraction of various information through parsing its tags.

Subsequently, in the second step, the Python application saves the evaluation results of each model in a Comma-Separated Values (CSV) file.

The third step consists of using a second Python application to analyse all the results stored on the CSV file and, as a final step, identify the most common defects present in these models.

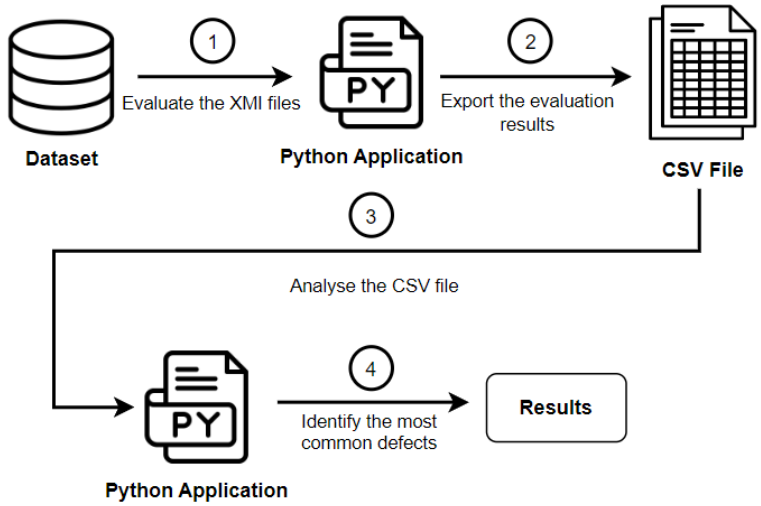


Figure 6.1: Overall evaluation process.

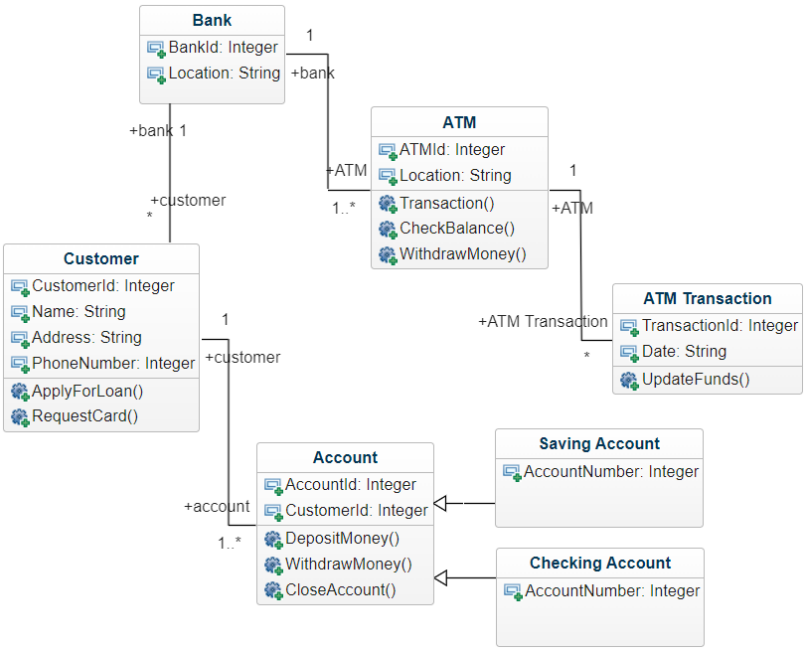


Figure 6.2: UML class diagram example.

Listing 6.1: Selected portion of XMI code representing the diagram in Figure 6.2.

```
<?xml version="1.0" encoding="UTF-8"?>
  <uml:Model xmi:id="_XmuAUBBHEe6q9cX5AkHPSQ" name="Bank Model">
  (...)
    <packagedElement xsi:type="uml:Class" xmi:id="_e2wvMDVREDywFcmCNq4K0w" name="Bank">
      <xmi:Extension extender="http://www.eclipse.org/emf/2002/Ecore">
        <eAnnotations xmi:id="_e2wvMDVREDywFcmCNq4K0w0" source="genmymodel">
          <details xmi:id="_e2wvMDVREDywFcmCNq4K0w00" key="uuid" value="
              ↪ _e2wvMDVREDywFcmCNq4K0w"/>
        </eAnnotations>
      </xmi:Extension>
      <ownedAttribute xmi:id="_lW4NQDVREDywFcmCNq4K0w" name="BankId" visibility="public">
        <xmi:Extension extender="http://www.eclipse.org/emf/2002/Ecore">
          <eAnnotations xmi:id="_lW4NQDVREDywFcmCNq4K0w0" source="genmymodel">
            <details xmi:id="_lW4NQDVREDywFcmCNq4K0w00" key="uuid" value="
                ↪ _lW4NQDVREDywFcmCNq4K0w"/>
          </eAnnotations>
        </xmi:Extension>
        <type xsi:type="uml:PrimitiveType" href="http://www.omg.org/spec/UML/20131001/
            ↪ PrimitiveTypes.xmi#//Integer"/>
      </ownedAttribute>
      <ownedAttribute xmi:id="_nMH3oDVREDywFcmCNq4K0w" name="Location" visibility="public">
        <xmi:Extension extender="http://www.eclipse.org/emf/2002/Ecore">
          <eAnnotations xmi:id="_nMH3oDVREDywFcmCNq4K0w0" source="genmymodel">
            <details xmi:id="_nMH3oDVREDywFcmCNq4K0w00" key="uuid" value="
                ↪ _nMH3oDVREDywFcmCNq4K0w"/>
          </eAnnotations>
        </xmi:Extension>
        <type xsi:type="uml:PrimitiveType" href="http://www.omg.org/spec/UML/20131001/
            ↪ PrimitiveTypes.xmi#//String"/>
      </ownedAttribute>
    </packagedElement>
    (...)
  </uml:Model>
</xmi:XMI>
```

7

Implementation

This chapter presents the implementation of the evaluation tool employed in this study. It provides an in-depth explanation of the implementation of each principle used in the automated evaluation process, accompanied by relevant code snippets. The complete source code of the evaluation tool is available in the following GitHub repository link.

7.1 Evaluation Tool

To evaluate UML class diagram models for the defects they contain, we developed an application in Python. This application consists of two files, namely, `main.py` and `utils.py`. The `main.py` file has the `main` function of the application and is the executable file (Listing 7.1). The `utils.py` file contains the helper functions that are called by the `main` function. Figure 7.1 illustrates the evaluation tool execution process.

To initialize this tool, it is necessary to specify the starting and ending index of the dataset files (`INITIAL_FILE` and `FINAL_FILE`) to establish a range of files that will be evaluated by this tool. Note that the index of the files is determined based on their extraction order, which was logged in a text file called `downloaded_files.txt`. To run this tool, the following command format must be used: `python main.py INITIAL_FILE FINAL_FILE`. For example, to process files from index 1 to 50000, the corresponding command is as follows: `python main.py 1 50000`.

Once the file range is defined, the program uses the `read_files_ids` function to read the filenames of the `downloaded_files.txt` file contained in that range. Then, one file is loaded at a time using the `read_xmi_file` function. This function uses the `ElementTree` library [Ele] to load the XMI file and represent it as a tree. And it returns the root node from this tree.

Listing 7.1: Evaluation tool main function code.

```
1 def main():
2
3     # Ensure the correct use
4     if len(sys.argv) != 3:
5         sys.exit("Usage: python main.py INITIAL_FILE FINAL_FILE")
```

```
6
7       INITIAL_FILE = int(sys.argv[1])
8       FINAL_FILE = int(sys.argv[2])
9
10      # Ensure that only valid numbers are used
11      if INITIAL_FILE < 1:
12          sys.exit("The initial file must be greater than 0!")
13
14      # Check if the end position of the file is less than the start position of the file
15      if FINAL_FILE < INITIAL_FILE:
16          sys.exit("The final file must be greater or equal than the initial file!")
17
18      # File range
19      files = utils.read_files_ids(INITIAL_FILE-1, FINAL_FILE)
20
21      # Evaluate each file
22      for file in files:
23
24          # Read the XMI file
25          FILE = file
26          root = utils.read_xmi_file(FOLDER + FILE)
27
28          # Evaluate the diagram using the Visual Expressiveness principle
29          elements_without_color = utils.find_elements_without_color(root)
30
31          # Evaluate the diagram using the Dual Coding principle
32          relationships_without_description = utils.find_relationships_without_description(
                ↪ root)
33
34          # Evaluate the diagram using the Graphic Economy principle
35          element_types = utils.check_number_of_different_element_types(root)
36
37          # Evaluate the diagram using the Perceptual Discriminability principle
38          perceptual_discriminability = utils.check_for_perceptual_discriminability(root)
39
40          # Evaluate the diagram using the Diagram Size principle
41          diagram_size = utils.check_for_diagram_size(root)
42
43          # Evaluate the diagram using the Overlapping Elements (Diagram Flaw) principle
44          overlapping_elements = utils.check_for_overlapping_elements(root)
45
46          # Evaluate the diagram using the Intersections (Diagram Flaw) principle
47          intersections = utils.check_for_intersections(root)
48
49          # Evaluate the diagram using the Bends in Lines (Diagram Flaw) principle
50          bends_in_lines = utils.check_for_bends_in_lines(root)
51
52          # Evaluate the diagram using the Close Lines (Diagram Flaw) principle
53          close_merged_or_aligned_lines = utils.check_for_close_merged_or_aligned_lines(root)
54
```

```
55      utils.save_results_to_csv(CSV_FILE + ".csv", FILE, elements_without_color,
            ↪ relationships_without_description, element_types,
            ↪ perceptual_discriminability, diagram_size, overlapping_elements,
            ↪ intersections, bends_in_lines, close_merged_or_aligned_lines)
```

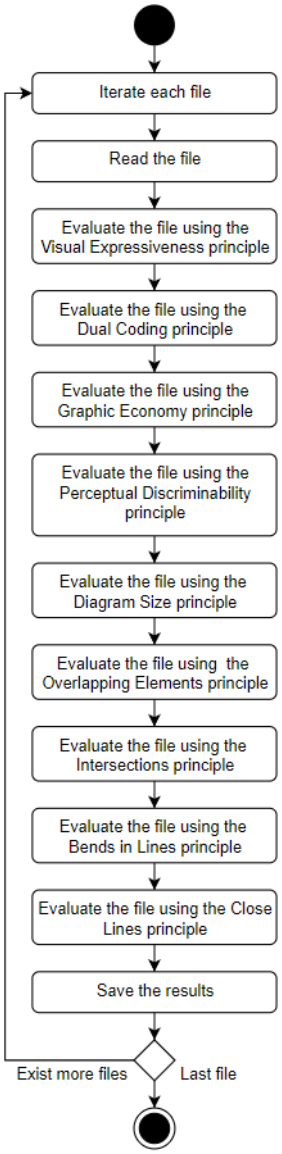


Figure 7.1: Activity diagram of the evaluation tool execution process.

The program uses the function `find_elements_without_color` (Listing 7.2) to analyze each file in the context of the Visual Expressiveness principle (described in subsection 5.1.1). This function receives the root node of an XMI file, represented by a tree and aims to identify elements within the file that do not have the `colour` attribute, i.e., that have the default colour.

To achieve this, the function employs the `findall` function from the `ElementTree` library to search for all elements with the `ownedDiagramElements` tags. These elements are then verified to determine if they correspond to UML class diagram elements. This

verification is performed by comparing the element's type attribute, specifically the attribute defined as `{http://www.w3.org/2001/XMLSchema-instance}type`, against an array of valid element types, denoted as `ELEMENTS_TYPE`.

Once the program identifies the UML class diagram elements, it proceeds to inspect each element individually to ascertain whether it possesses a `colour` attribute. If an element lacks the `colour` attribute, the function captures its ID and stores it in a set referred to as `elements_without_color`.

Finally, the function returns `True` if the diagram does not respect this principle, on the contrary, it returns `False`. Additionally, it returns the number of UML class diagram elements in the XMI file that do not possess a `colour` attribute.

Listing 7.2: `find_elements_without_color` function code.

```
ELEMENTS_TYPE = ["PackageWidget", "ClassWidget", "InterfaceWidget", "DataTypeWidget", "
    ↪ EnumerationWidget", "GeneralizationSegment", "InterfaceRealizationSegment", "
    ↪ DependencySegment", "UsageSegment", "ImportSegment", "TemplateBindingSegment", "
    ↪ AssociationSegment", "InnerElementSegment"]

def find_elements_without_color(root):

    elements_without_color = set()

    # Go through all elements of the diagram
    for elem in root.findall(".//ownedDiagramElements"):

        # Check if it is a UML class diagram element
        if elem.attrib['{http://www.w3.org/2001/XMLSchema-instance}type'] in ["com.
            ↪ genmymodel.graphic.uml:" + elem_type for elem_type in ELEMENTS_TYPE]:

            # Check if the element does not have the color attribute
            if 'color' not in elem.attrib and 'modelElement' in elem.attrib:
                id = elem.attrib['modelElement']
                elements_without_color.add(id)

    return [len(elements_without_color) > 0, len(elements_without_color)]
```

To evaluate the Dual Coding principle (described in subsection 5.1.2), the program employs a function called `find_relationships_without_description` (Listing 7.3) to analyze each file. This function aims to identify relationships within the UML class diagrams without a name or a multiplicity (if applicable).

The function uses the `findall` function from the `ElementTree` library to search for all elements with the `packagedElement` tags that contain the attribute type equal to `uml:Dependency`, `uml:Usage`, `uml:Association`, or `uml:AssociationClass`. These attribute types represent the different types of relationships within a diagram, excluding generalizations, realizations, imports, template bindings and inner relations, which do not contain names or multiplicities.

After identifying these elements, the function proceeds to examine each relationship individually. For the `uml:Dependency` and `uml:Usage` types, it checks if the relationships do not have the `name` attribute or if it is empty. If any of these conditions are met, the `relationships_without_description` counter is incremented, indicating the presence of a relationship without a name.

For the `uml:Association` and `uml:AssociationClass` types, it retrieves the elements that comprise each relationship, again using the `findall` function, which targets the `ownedEnd` tags. These `ownedEnd` elements represent the connected ends of the relationship. The program then verifies whether each element within the relationship has a name. This verification is done by checking for the presence of the `name` attribute within each `ownedEnd` element or if it is empty. Additionally, the `uml:Association` type is checked to see if each element within the relationship has a multiplicity. This is achieved by checking for the presence of the `value` attributes in the `lowerValue` and `upperValue` tags within each `ownedEnd` element, or if they are empty. If any element does not contain these attributes or they are empty, it signifies the absence of a name or a multiplicity for that specific relationship. In that case, the `relationships_without_description` counter is incremented.

Finally, the function returns `True` if the diagram contains one or more relationships without a name or a multiplicity, otherwise, it returns `False`. Additionally, it returns the number of relationships without a name or a multiplicity.

Listing 7.3: `find_relationships_without_description` function code.

```
1  def find_relationships_without_description(root):
2
3      relationships_without_description = 0
4
5      # Check the Dependency type relationships
6      for dependency in root.findall(".//packagedElement[@{http://www.w3.org/2001/XMLSchema-
           ↪ instance}type='uml:Dependency']"):
7          if 'name' not in dependency.attrib or dependency.attrib["name"] == "":
8              relationships_without_description += 1
9
10     # Check the Usage type relationships
11     for usage in root.findall(".//packagedElement[@{http://www.w3.org/2001/XMLSchema-
           ↪ instance}type='uml:Usage']"):
12         if 'name' not in usage.attrib or usage.attrib["name"] == "":
13             relationships_without_description += 1
14
15     # Check the Association / Composition / Aggregation type relationships
16     for association in root.findall(".//packagedElement[@{http://www.w3.org/2001/XMLSchema-
           ↪ instance}type='uml:Association']"):
17         owned_ends = association.findall("./ownedEnd")
18
19         for owned_end in owned_ends:
20             if 'name' not in owned_end.attrib or owned_end.attrib["name"] == "":
21                 relationships_without_description += 1
```

```
            break

        lower_value = owned_end.find("./lowerValue")
        upper_value = owned_end.find("./upperValue")

        if 'value' not in lower_value.attrib or lower_value.attrib["value"] == "" or '
            ↪ value' not in upper_value.attrib or upper_value.attrib["value"] == "":
            relationships_without_description += 1
            break

    # Check the Association Class type relationships
    for association_class in root.findall(".//packagedElement[@{http://www.w3.org/2001/
        ↪ XMLSchema-instance}type='uml:AssociationClass']"):
        owned_ends = association_class.findall("./ownedEnd")

        for owned_end in owned_ends:
            if 'name' not in owned_end.attrib or owned_end.attrib["name"] == "":
                relationships_without_description += 1
                break

    return [relationships_without_description > 0, relationships_without_description]
```

To evaluate the Graphic Economy principle (described in subsection 5.1.3), the program uses the `check_number_of_ different_element_types` function (Listing 7.4) to analyze each file. The objective of this function is to determine whether the number of different element types present in a UML class diagram exceeds six, thereby evaluating the graphical economy of the diagram.

The `check_number_of_different_element_types` function uses the `findall` function from the `ElementTree` library to search for all elements with the `ownedDiagramElements` tag. These elements are then verified to determine if they correspond to UML class diagram elements. This verification is done by comparing the element's type attribute, concretely the attribute defined as `{http://www.w3.org/2001/XMLSchema-instance}type`, against an array of valid element types, denoted as `ELEMENTS_TYPE`.

Once the program identifies the UML class diagram elements, it adds each element type to the set called `element_types`, thus storing only different element types.

Finally, the function returns the number of different element types in the diagram and whether this number surpasses six or not, returning `True` or `False`, respectively.

Listing 7.4: `check_number_of_different_element_types` function code.

```
def check_number_of_different_element_types(root):

    element_types = set()

    # Iterate through all elements of the diagram
    for elem in root.findall(".//ownedDiagramElements"):

        # Check if it is a UML class diagram element
```

```
9         if elem.attrib['{http://www.w3.org/2001/XMLSchema-instance}type'] in ["com.
              ↪ genmymodel.graphic.uml:" + elem_type for elem_type in ELEMENTS_TYPE]:
10
11            # Add the element type to the set
12            element_types.add(elem.attrib['{http://www.w3.org/2001/XMLSchema-instance}type'
                  ↪ ])
13
14    return [len(element_types) > 6, len(element_types)]
```

To evaluate the Perceptual Discriminability principle (described in subsection 5.1.4), the program uses the `check_for_perceptual_discriminability` function (Listing 7.5) to analyze each file. This function aims to determine the distinguishability of elements within a UML class diagram, specifically by examining their types, names, colours, attributes, operations, and enumeration literals (if applicable).

The function utilizes the `findall` function from the `ElementTree` library to retrieve all elements within the diagram. It then proceeds to construct a dictionary called `type_dict`, which organizes the relevant information for each element type. This information includes the element's ID, name, colour, attributes, operations, and enumeration literals.

For each element found, the function extracts its attributes and adds them to the corresponding type entry within `type_dict`. The element's type is determined by parsing the `{http://www.w3.org/2001/XMLSchema-instance}type` attribute and extracting the type name.

After the dictionary is populated, the function performs two evaluations. Firstly, it checks if all elements in the diagram belong to distinct types. If this is the case, it concludes that the elements are perceptually discriminable, indicating that they are visually distinguishable from one another. In this scenario, the function returns `True` and a count of 0, meaning that there are no occurrences of identical elements.

However, if there are elements with the same type, the function proceeds to evaluate their discriminability. It compares the attributes (colour, name, attributes, operations, and enumeration literals) of each pair of elements within the same type. Note that it performs specific attribute comparisons based on the element types, taking into account different attributes for different element types. The function identifies occurrences where these elements are identical, indicating a lack of perceptual discriminability. The function maintains a counter, `occurrences`, to track the number of instances where elements within the same type are indistinguishable.

After evaluating all element types, the function checks the value of `occurrences`. If it is greater than zero, it concludes that there are occurrences of indistinguishable elements within the diagram. In this case, it returns `False` and the count of occurrences. Conversely, if occurrences are zero, the function determines that all elements within the diagram are perceptually discriminable and return `True` with a count of 0.

Listing 7.5: `check_for_perceptual_discriminability` function code.

```
1 def check_for_perceptual_discriminability(root):
```

```
2
3       ELEMENT_TYPES_IN_DIAGRAM = set()
4
5       # Get all elements from the diagram
6       diagram_elements = root.findall(".//*[@{http://www.w3.org/2001/XMLSchema-instance}type
            ↪ ]")
7
8       # Create a dictionary to store information about each element type
9       type_dict = {elem_type: {"ids": [], "names": [], "colors": [], "attributes": [], "
            ↪ operations": [], "enum_literals": []} for elem_type in ELEMENTS_TYPE}
10
11      # Iterate through all elements of the diagram
12      for element in diagram_elements:
13          elem_type = element.get("{http://www.w3.org/2001/XMLSchema-instance}type").split(':
                ↪ ')[-1]
14
15          # Check if the element exists in the dictionary
16          if elem_type in type_dict:
17
18              # Save the element type in the set
19              ELEMENT_TYPES_IN_DIAGRAM.add(elem_type)
20
21              # Extract the element's ID, name, colour, attributes, operations and
                    ↪ enumeration literals (if it exists)
22              elem_id = element.attrib.get('modelElement')
23
24              if elem_id is None:
25                  continue
26
27              elem_name = ""
28              elem_color = ""
29              elem_attributes = []
30              elem_operations = []
31              elem_enum_literals = []
32
33              if elem_type in ELEMENTS:
34                  elem_element = root.find(".//*[@{http://schema.omg.org/spec/XMI/2.1}id='" +
                        ↪ elem_id + "']")
35
36                  if elem_element is not None:
37                      elem_name = elem_element.attrib.get('name', '')
38
39              if 'color' in element.attrib:
40                  elem_color = element.attrib['color']
41
42              if elem_type in {"ClassWidget", "InterfaceWidget"}:
43                  elem_attributes = [attribute.attrib.get('name', '') for attribute in root.
                        ↪ findall(".//*[@{http://schema.omg.org/spec/XMI/2.1}id='" + elem_id +
                        ↪  "']/ownedAttribute")]
```

```
44              elem_operations = [operation.attrib.get('name', '') for operation in root.
                    ↪ findall(".//*[@{http://schema.omg.org/spec/XMI/2.1}id='" + elem_id +
                    ↪  "']/ownedOperation")]
45
46          if elem_type == "EnumerationWidget":
47              elem_enum_literals = [enum_literal.attrib.get('name', '') for enum_literal
                    ↪ in root.findall(".//*[@{http://schema.omg.org/spec/XMI/2.1}id='" +
                    ↪ elem_id + "']/ownedLiteral")]
48
49          # # Add the element's ID, name and colour, attributes, operations and
                ↪ enumeration literals to the dictionary
50          type_dict[elem_type]["ids"].append(elem_id)
51          type_dict[elem_type]["names"].append(elem_name)
52          type_dict[elem_type]["colors"].append(elem_color)
53          type_dict[elem_type]["attributes"].append(elem_attributes)
54          type_dict[elem_type]["operations"].append(elem_operations)
55          type_dict[elem_type]["enum_literals"].append(elem_enum_literals)
56
57  # Check if all elements are of different types
58  if all(len(type_dict[elem_type]["ids"]) == 1 for elem_type in ELEMENTS_TYPE):
59      return [True, 0]
60
61  # Check whether elements of each type are perceptually discriminable
62  else:
63      occurrences = 0
64
65      for elem_type in ELEMENT_TYPES_IN_DIAGRAM:
66          ids = type_dict[elem_type]["ids"]
67          colors = type_dict[elem_type]["colors"]
68          names = type_dict[elem_type]["names"]
69          attributes = type_dict[elem_type]["attributes"]
70          operations = type_dict[elem_type]["operations"]
71          enum_literals = type_dict[elem_type]["enum_literals"]
72
73          # Check whether elements are perceptually discriminable
74
75          if elem_type in {"ClassWidget", "InterfaceWidget"}:
76              for i in range(len(ids)):
77                  for j in range(i + 1, len(ids)):
78                      if colors[i] == colors[j] and names[i] == names[j] and attributes[i]
                            ↪ == attributes[j] and operations[i] == operations[j]:
79                          occurrences += 1
80
81          elif elem_type in {"PackageWidget", "DataTypeWidget"}:
82              for i in range(len(ids)):
83                  for j in range(i + 1, len(ids)):
84                      if colors[i] == colors[j] and names[i] == names[j]:
85                          occurrences += 1
86
87          elif elem_type == "EnumerationWidget":
```

```
88              for i in range(len(ids)):
89                  for j in range(i + 1, len(ids)):
90                      if colors[i] == colors[j] and names[i] == names[j] and enum_literals[
                            ↪ i] == enum_literals[j]:
91                          occurrences += 1
92
93          else:
94              for i in range(len(ids)):
95                  for j in range(i + 1, len(ids)):
96                      if colors[i] == colors[j] and colors[i]:
97                          occurrences += 1
98
99      if occurrences > 0:
100         return [False, occurrences]
101     else:
102         return [True, 0]
```

To evaluate the Diagram Size principle, the program uses the `check_for_diagram_size` function (Listing 7.6) to analyze each file. The objective of this function is to determine whether the number of elements in a UML class diagram falls within the range of 20 to 60 elements, as defined by Störrle [Stö14].

To achieve this, the function employs the `findall` function from the `ElementTree` library to search for all elements with the `ownedDiagramElements` tags. These elements are then verified to determine if they correspond to UML class diagram elements. This verification process is performed by comparing the element's type attribute (i.e., the `http://www.w3.org/2001/XMLSchema-instancetype` attribute) with an array of valid element types, called `ELEMENTS_TYPE`. If an element is a UML class diagram element, the `diagrams_elements` counter is incremented.

Finally, the function returns `True` if the number of elements in the diagram is within the range of 20 to 60 elements, otherwise, it returns `False`. Additionally, it returns the number of elements in the diagram.

Listing 7.6: `check_for_diagram_size` function code.

```
1  def check_for_diagram_size(root):
2
3      diagram_elements = 0
4
5      # Iterate through all diagram elements
6      for elem in root.findall(".//ownedDiagramElements"):
7
8          # Check if it is a UML class diagram element
9          if 'modelElement' in elem.attrib and elem.attrib['{http://www.w3.org/2001/XMLSchema
               ↪ -instance}type'] in ["com.genmymodel.graphic.uml:" + elem_type for
               ↪ elem_type in ELEMENTS_TYPE]:
10             diagram_elements += 1
11
12     return [diagram_elements >= 20 and diagram_elements <= 60, diagram_elements]
```

To evaluate the Overlapping Elements principle, the program uses the `check_for_overlapping_elements` function (Listing 7.7) to analyze each file. This function aims to identify the elements within a UML class diagram that overlap each other spatially.

To accomplish this, the function employs an auxiliary function called `get_elements_info`. This function retrieves relevant information about each element in the diagram, specifically their IDs, as well as their positional coordinates (x, y) and dimensions (width and height).

Once the element information is obtained, the function proceeds to compare each element against all other elements within the diagram. It checks whether there is any spatial overlap between two elements by examining their positional coordinates and dimensions. The function employs a nested loop structure to compare each pair of elements only once. It calculates the boundary coordinates (x, y, width, and height) for both elements and checks if they intersect on both the x-axis and y-axis. If an overlap is detected, the function increments a counter, `overlapping_elements`, used to track the number of occurrences.

Upon completion of the comparison process, the function evaluates the value of the `overlapping_elements` counter. If it is greater than zero, it concludes that there are overlapping elements within the diagram. In this case, it returns `True` along with the count of `overlapping_elements`. Conversely, if `overlapping_elements` is zero, the function determines that there are no overlapping elements and returns `False` with a count of 0.

Listing 7.7: `check_for_overlapping_elements` function code.

```
def check_for_overlapping_elements(root):

    # Get x, y, width and height of each element
    elements = get_elements_info(root)

    overlapping_elements = 0

    # Iterate through the elements
    for i, (id_1, pos1) in enumerate(elements.items()):
        for id_2, pos2 in list(elements.items())[i+1:]:
            x1, y1, w1, h1 = pos1
            x2, y2, w2, h2 = pos2

            # Check if elements overlap
            if x1 < x2 + w2 and x1 + w1 > x2:
                if y1 < y2 + h2 and y1 + h1 > y2:
                    overlapping_elements += 1

    return [overlapping_elements > 0, overlapping_elements]
```

To evaluate the Intersections principle, the program uses the `check_for_intersections` function (Listing 7.8) to analyze each file. Its objective is to identify intersections between relationships or relationships and elements in a UML class diagram.

First, this function employs an auxiliary function called `get_relationships_elements_coordinates`. This function retrieves the coordinates of each relationship's elements. It collects information about the initial, intermediate, and final points, of each relationship, within the diagram.

Upon obtaining the relationship element coordinates, the function proceeds to break down the relationships into line segments and store these line segments in an array called `line_segments`.

To identify intersections between line segments, the function employs a nested loop structure. Each line segment is compared against all other line segments using the `has_intersection` auxiliary function. This function determines if two line segments intersect by evaluating the orientations of the respective points. If an intersection is detected, the function increments the `intersections` counter. Note that the points of connection of each line segment of a relationship are disregarded, so they do not count as intersections.

Furthermore, the function examines intersections between relationships and other diagram elements. To achieve this, the `get_elements_info` auxiliary function is used. This function retrieves essential information about each element in the diagram, such as their IDs, positional coordinates (x, y), and dimensions (width and height). The line segments are then tested against the boundaries of each element using the `has_intersection function`. If an intersection occurs, the `intersections` counter is incremented.

Finally, the function returns `True` or `False` if there are intersections or not, and also the number of intersections.

Listing 7.8: `check_for_intersections` function code.

```
def check_for_intersections(root):

    # Get the coordinates of the elements of each relation
    relationships = get_relationships_elements_coordinates(root)

    line_segments = []
    intersections = 0

    # Get the line segments of each relationship
    for id, points in relationships.items():

        # Check if the intermediate points are different by more than 0 units
        if abs(points[1][0] - points[2][0]) > 0 or abs(points[1][1] - points[2][1]) > 0:
            line_segments.append([points[0], points[1]])
            line_segments.append([points[1], points[2]])
            line_segments.append([points[2], points[3]])

        # Otherwise, consider only the starting and ending points, that is, the
            ↪ relationship is a straight line
        else:
            line_segments.append([points[0], points[3]])
```

```
21
22      # Check if there are intersections between line segments
23      for i in range(len(line_segments)):
24          for j in range(i + 1, len(line_segments)):
25              if has_intersection(line_segments[i][0], line_segments[i][1], line_segments[j
                    ↪ ][0], line_segments[j][1]):
26                  intersections += 1
27
28      # Get x, y, width and height of each element
29      elements = get_elements_info(root)
30
31      # Check if there are intersections between relations and other elements
32      for i in range(len(line_segments)):
33          for elem_id, elem_info in elements.items():
34              x = elem_info[0]
35              y = elem_info[1]
36              width = elem_info[2]
37              height = elem_info[3]
38
39              # Check if the line segments intersects the element
40              if has_intersection(line_segments[i][0], line_segments[i][1], (x, y), (x +
                    ↪ width, y)) or \
41                 has_intersection(line_segments[i][0], line_segments[i][1], (x, y), (x, y +
                    ↪ height)) or \
42                 has_intersection(line_segments[i][0], line_segments[i][1], (x + width, y), (x
                    ↪  + width, y + height)) or \
43                 has_intersection(line_segments[i][0], line_segments[i][1], (x, y + height), (
                    ↪ x + width, y + height)):
44                  intersections += 1
45
46      return [intersections > 0, intersections]
```

To evaluate the Bends in Lines principle, the program uses the `check_for_bends_in_lines` function (Listing 7.9) to analyze each file. This function aims to identify any bends present in the lines of relationships within UML class diagrams.

First, the function begins by calling the `get_relationships_elements_coordinates` auxiliary function. This function retrieves the coordinates of each relationship's elements, specifically the initial point, the two intermediate points, and the final point. Note that the relationships created in GenMyModel always contain two intermediate points. Therefore, relationships can have a maximum of two bends in their lines.

Next, for each relationship, the function examines whether the relationship does not represent a straight line, indicating the presence of bends. To accomplish this, the function checks whether the intermediate points differ by more than zero units in either the horizontal or vertical direction. If the intermediate points satisfy this condition, the function utilizes the `has_two_lines` auxiliary function to determine if the relationship exhibits two lines, suggesting the presence of a single bend. Alternatively, if the relationship does not has two lines, it is classified as having three lines, indicating the presence of two bends.

Finally, the function returns `True` or `False` if there are bends in the lines and the total count of bends identified.

Listing 7.9: `check_for_bends_in_lines` function code.

```
def check_for_bends_in_lines(root):

    bend_in_lines = 0

    # Get the coordinates of the elements of each relation
    relationships = get_relationships_elements_coordinates(root)

    for id, points in relationships.items():

        # Check if the relationship has bends
        if abs(points[1][0] - points[2][0]) > 0 or abs(points[1][1] - points[2][1]) > 0:

            # Check if has one bend
            if has_two_lines(points[0], points[1], points[2], points[3]):
                bend_in_lines += 1

            # Check if has two bends
            else:
                bend_in_lines += 2

    return [bend_in_lines > 0, bend_in_lines]
```

To evaluate the Close Merged or Aligned Lines principle, the program uses the `check_for_close_merged_or_aligned_lines` function (Listing 7.10) to analyze each file. This function aims to identify merged or closely positioned relationships, excluding cases where relationships are of the same type and share exactly one endpoint.

The function begins by calling the `get_relationships_elements_coordinates` auxiliary function. This function retrieves the coordinates of each association's elements, namely the initial, intermediate, and final points of each relationship within the diagram.

Having obtained the relationships element coordinates, the function proceeds to break down the relationships into line segments, storing them in an array called `line_segments`.

Subsequently, the function examines each pair of line segments to determine if they are closely positioned using the `is_close` auxiliary function. Note that line segments from the same relationship are disregarded.

Additionally, it checks whether the relationships share exactly one endpoint using the `share_one_endpoint` auxiliary function and checks whether the relationships have the same type using the `same_relationship_type` auxiliary function. If the line segments are close, do not share an endpoint and have different types, the `close_lines` counter is incremented.

Finally, the function returns a `True` or `False`, indicating whether there are any close lines that do not share an endpoint and have different types, along with the total count of

identified closed lines.

Listing 7.10: `check_for_close_merged_or_aligned_lines` function code.

```
def check_for_close_merged_or_aligned_lines(root):
    close_lines = 0
    line_segments = []

    # Get the coordinates of the elements of each relation
    relationships = get_relationships_elements_coordinates(root)

    # Get the line segments of each relationship
    for id, points in relationships.items():
        if abs(points[1][0] - points[2][0]) > 0 or abs(points[1][1] - points[2][1]) > 0:
            line_segments.append([id, [points[0], points[1]]])
            line_segments.append([id, [points[1], points[2]]])
            line_segments.append([id, [points[2], points[3]]])
        else:
            line_segments.append([id, [points[0], points[3]]])

    # Check if there are merged or aligned lines that are close together and do not share
        ↪ one endpoint and do not have the same type
    for i in range(len(line_segments)):
        for j in range(i + 1, len(line_segments)):

            # Check if the line segements are not from the same relationship
            if line_segments[i][0] != line_segments[j][0]:

                if is_close(line_segments[i][1], line_segments[j][1]) and not
                    ↪ share_one_endpoint(root, line_segments[i][0], line_segments[j][0])
                    ↪ and not same_relationship_type(root, line_segments[i][0],
                    ↪ line_segments[j][0]):
                    close_lines += 1

    return [close_lines > 0, close_lines]
```

The last step is to save the previous results obtained by evaluating each diagram using the functions described, in a CSV file called `result.csv`, using the `save_results_to_csv` function (Listing 7.11). In this file is stored for each row the filename and the corresponding results.

Listing 7.11: `save_results_to_csv` function code.

```
def save_results_to_csv(filename, file_id, elements_without_color,
    ↪ associations_without_description, element_types, perceptual_discriminability,
    ↪ overlapping_elements, intersections, bends_in_lines, close_merged_or_aligned_lines
    ↪ ):

    # Open the text file
    with open(filename, 'a', newline='') as file:
        writer = csv.writer(file)
```

```
6
7       # Save the results
8       writer.writerow([file_id, elements_without_color, relationships_without_description
            ↪ , element_types, perceptual_discriminability, diagram_size,
            ↪ overlapping_elements, intersections, bends_in_lines,
            ↪ close_merged_or_aligned_lines])
```

8

Results

This chapter begins by presenting the general results derived from the analysis of the collected dataset. Additionally, it shows and describes the results obtained from running the automated evaluation tool on the dataset. Lastly, it presents a critical discussion of the results. The source code used to analyze the results is available in the following GitHub repository link.

8.1 General Results

The execution of the web scraping tool to scrape the public repositories of the GenMyModel platform resulted in the collection of a dataset comprising 103,103 UML class diagrams, all in the XMI format. This dataset serves as the basis for subsequent analysis and examination.

The initial phase of analysis focused on extracting information about the author of each file. For this, the metadata of each file was examined by searching the `details` tag possessing the `key='author'` attribute. We identified 28,138 distinct authors. Further, 56,246 diagrams were created by undefined authors, meaning that there wasn't any information in the metadata regarding the authors.

Subsequently, the distribution of diagrams was analysed based on the number of entities they encompass, as illustrated in Figure 8.1. Note that the entities consist of the following UML elements: package, class, interface, datatype and enumeration. The analysis revealed that the dataset predominantly consists of diagrams with 1 to 26 entities, representing 92.3% (95,213 diagrams) of the dataset. Among these, the most frequent are diagrams containing ten entities. Additionally, the distribution of the number of entities was analysed, as shown in Figure 8.2. This revealed that the median number of entities is 10, and the average number of entities is 16.19.

Furthermore, the distribution of diagrams was analysed based on the number of relationships they contain, as illustrated in Figure 8.3. Note that the relationships consist of the following UML elements: generalization, realization, dependency, usage, import, template binding, association, composition, aggregation, inner relation and association

class. The analysis showed that the dataset mainly contains diagrams with 1 to 22 relationships, representing 88.6% (91,394 diagrams) of the dataset. Within this group, five relationships are the most frequent in these diagrams. Additionally, the distribution of the number of relationships was analysed, as shown in Figure 8.4. This revealed that the median number of relationships is 10, and the average number of relationships is 14.61.

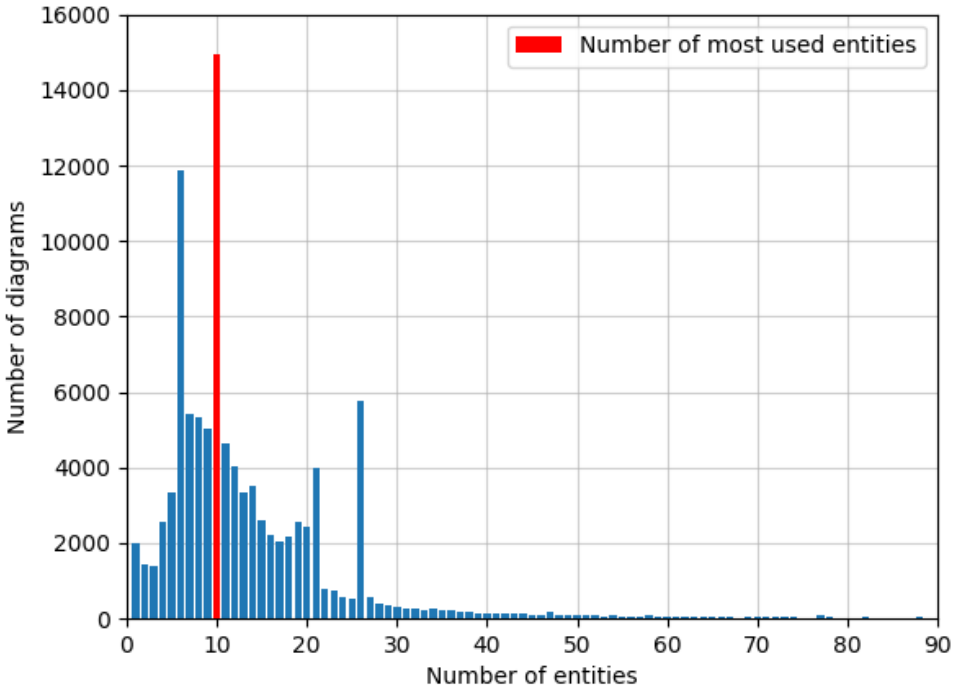


Figure 8.1: Distribution of diagrams based on their number of entities (with axes limited for clarity).

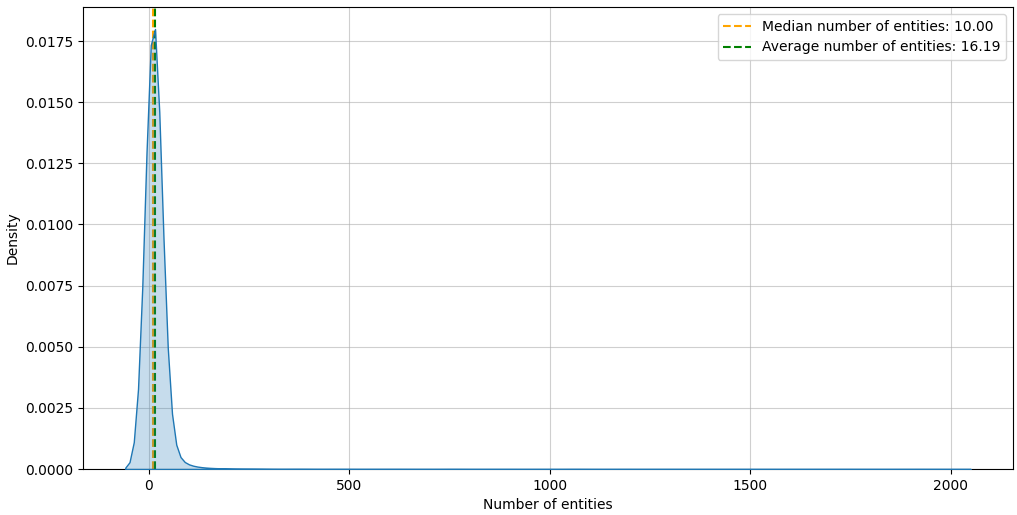


Figure 8.2: Density plot based on the number of entities.

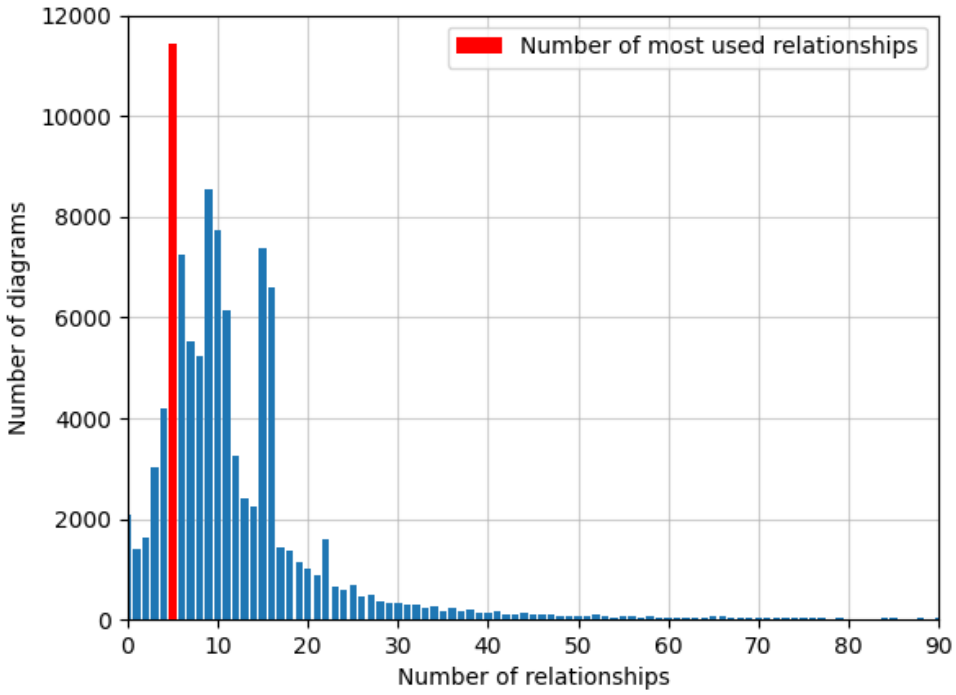


Figure 8.3: Distribution of diagrams based on their number of relationships (with axes limited for clarity).

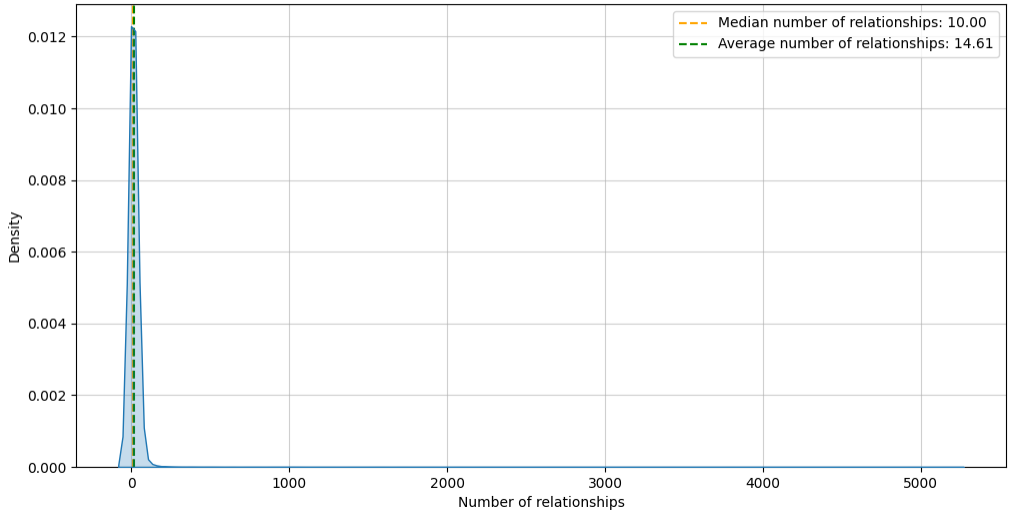


Figure 8.4: Density plot based on the number of relationships.

Also, the distribution of the UML class diagram elements in the diagrams of the dataset was examined, as shown in Figure 8.5. The results revealed that the most used entities are classes with attributes and operations. In addition, among the various types of relationships present in the diagrams, associations emerged as the most used, followed by generalizations.

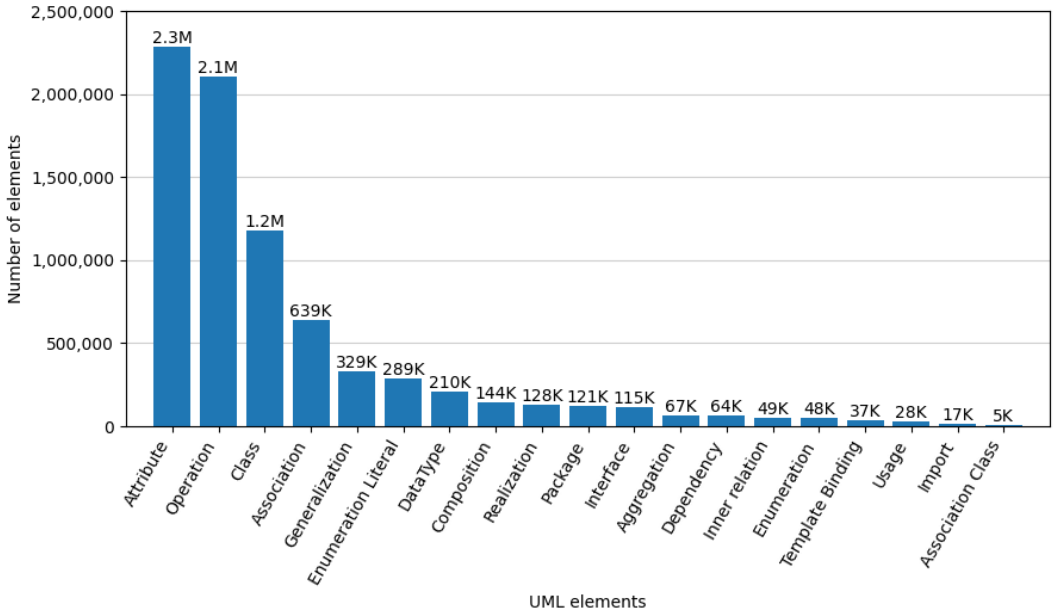


Figure 8.5: Distribution of the UML elements.

Lastly, the percentage of diagram usage within the XMI files was studied. Note that all files extracted from the GenMyModel platform contain a UML class diagram. However, each file can contain elements from different types of UML diagrams. This occurs because the GenMyModel editor allows mixed use of entities from component, deployment, object and use case diagrams, as shown in Figure 8.6.

Table 8.1 presents the percentage of diagram usage in the XMI files. The results show that the majority of models have exclusively a class diagram (78.294%), followed by models with a class and use case diagram (15.947%). The remaining models with other types are very residual, accounting for only 5.759%.

Table 8.1: Percentage of diagram usage.

Diagram type	**Number of models**	**Percentage**
Class Diagram	80723	78.294%
Class Diagram, Use Case Diagram	16442	15.947%
Class Diagram, Component Diagram, Use Case Diagram	2850	2.764%
Class Diagram, Object Diagram	1098	1.065%
Class Diagram, Component Diagram, Deployment Diagram, Use Case Diagram	482	0.467%
Class Diagram, Object Diagram, Use Case Diagram	374	0.363%

Class Diagram, Component Diagram, Object Diagram, Use Case Diagram	299	0.290%
Class Diagram, Component Diagram	295	0.286%
Class Diagram, Component Diagram, Deployment Diagram, Object Diagram, Use Case Diagram	194	0.188%
Class Diagram, Deployment Diagram, Use Case Diagram	130	0.126%
Class Diagram, Deployment Diagram	88	0.085%
Class Diagram, Component Diagram, Object Diagram	63	0.061%
Class Diagram, Component Diagram, Deployment Diagram	32	0.031%
Class Diagram, Deployment Diagram, Object Diagram	18	0.017%
Class Diagram, Component Diagram, Deployment Diagram, Object Diagram	12	0.011%
Class Diagram, Deployment Diagram, Object Diagram, Use Case Diagram	3	0.002%

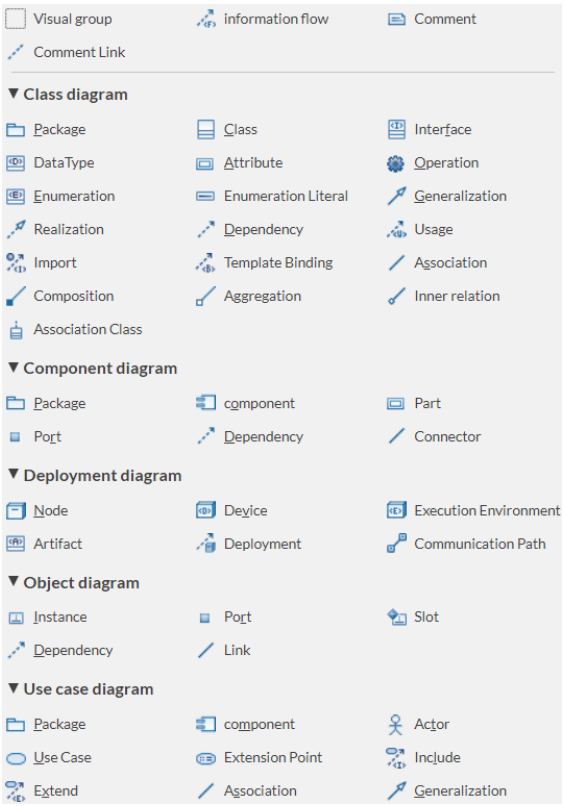


Figure 8.6: GenMyModel editor for UML class diagrams.

8.2 Evaluation Results

After conducting the evaluation using the tool detailed in Chapter 7, we proceeded to analyze the resulting data. The main objective of this study was to identify common defects present in the collected UML class diagrams. Finding these defects involved assessing whether the principles outlined in Chapter 5 were adhered to or violated.

Our first step was to analyze the total number of occurrences in the diagrams where the principles were not adhered to. Figure 8.7 shows the total number of occurrences of non-adherence to each principle. The analysis showed that the total number of occurrences for all principles not respected is 6,994,547 occurrences. Furthermore, the most common principles not respected are the Close Lines principle (with 2,221,340 occurrences), followed by the Visual Expressiveness principle (with 2,216,424 occurrences), the Intersections principle (with 1,287,512 occurrences) and the Overlapping Elements principle (with 612,577 occurrences), accounting for 90.7% of all occurrences. As expected, the Diagram Size and Graphic Economy principles are the ones with the lowest occurrences, as for these principles there can only be one occurrence per diagram.

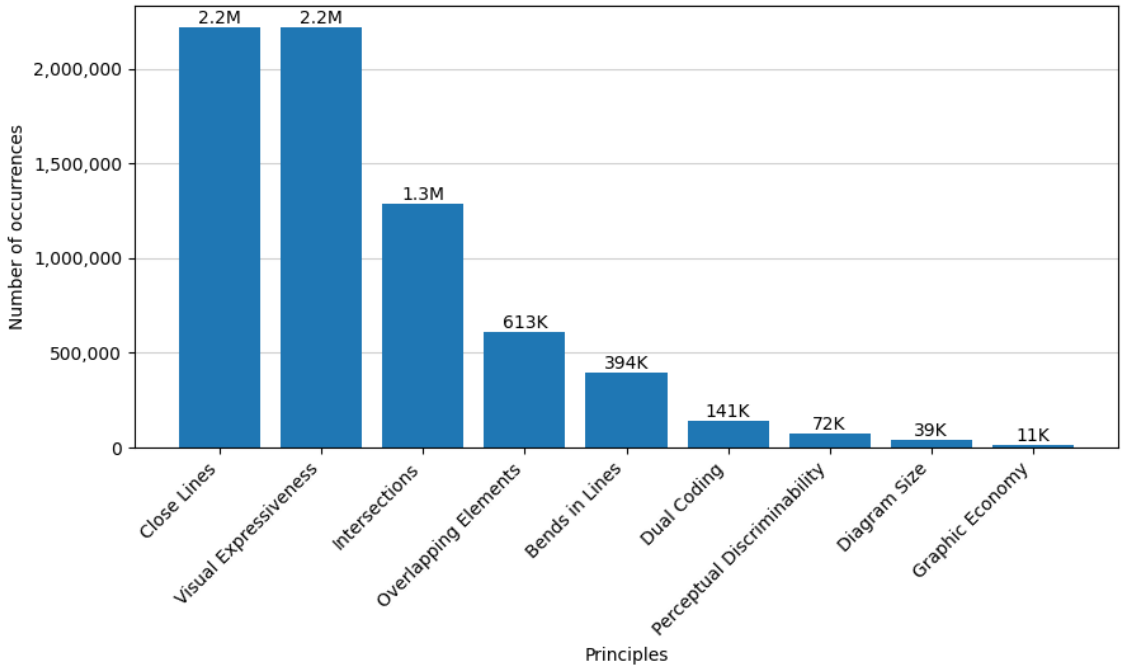


Figure 8.7: Total count of each principle not respected.

Subsequently, we studied the number of diagrams that exhibited at least one instance of not adhering to each principle. Addressing this concern was essential, as our previous examination focused solely on the total number of occurrences of non-compliance for each principle. Moreover, it became apparent that certain errors were persistently made, as modellers who disregarded a particular principle could do so repeatedly within the same diagram.

Figure 8.8 illustrates the number of diagrams in which each principle is not respected at least once. The analysis showed that the most frequently violated principles, with more than half of the dataset not complying with these principles, are the Visual Expressiveness principle (with 96,382 diagrams that do not respect this principle) and the Bends in Lines principle (with 51,520 diagrams that do not respect this principle).

Also, it is now possible to compare the Diagram Size and Graphic Economy principles with the other principles, as we are checking whether or not each diagram respects each principle, which is not possible with the total number of occurrences previously shown in Figure 8.7.

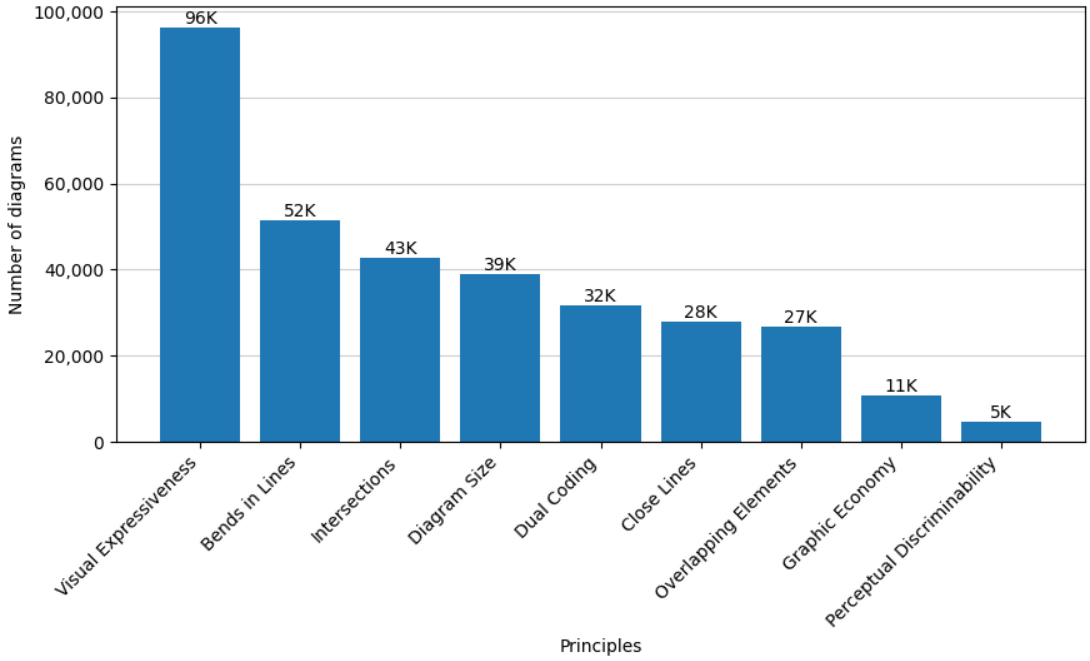


Figure 8.8: Count of each principle not respected at least once.

In more detail, Figures 8.9 to 8.17 show the percentage of diagram compliance with each principle. From this, we can divide the diagram compliance with each principle into three subsets. The first subset represents low compliance of the dataset with the principles (less than 10%) and includes the Visual Expressiveness principle, which is respected only by 6.5% of the dataset. The second subset represents moderate compliance of the dataset with the principles (between 50% and 70%) and includes the Bends in Lines principle (50.0%), the Intersections principle (58.4%), the Diagram Size principle (62.2%), and the Dual Coding principle (69.2%). The last subset represents high compliance of the dataset with the principles (more than 70%) and includes the Close Lines principle (73.0%), the Overlapping Elements principle (74.0%), the Graphic Economy principle (89.4%), and the Perceptual Discriminability principle (95.4%).

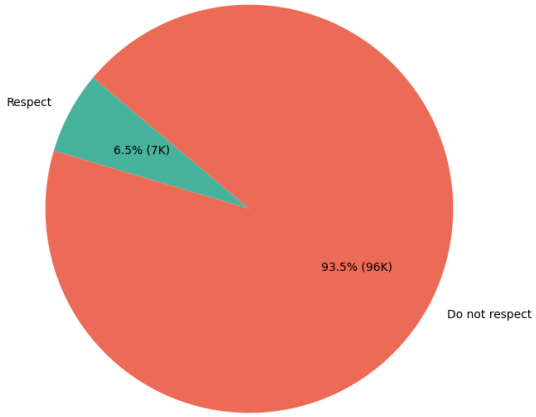


Figure 8.9: Diagram compliance with the Visual Expressiveness principle.

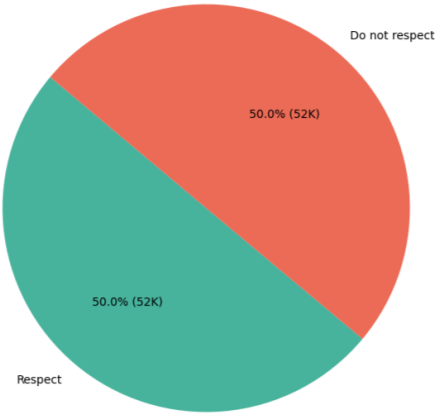


Figure 8.10: Diagram compliance with the Bends in Lines principle.

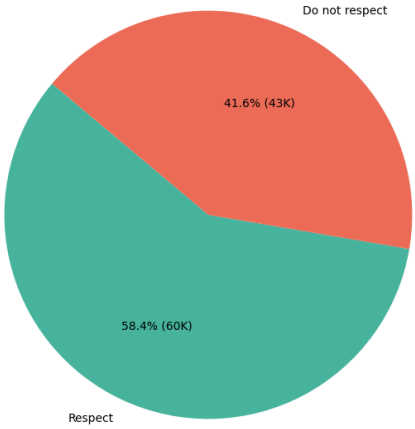


Figure 8.11: Diagram compliance with the Intersections principle.

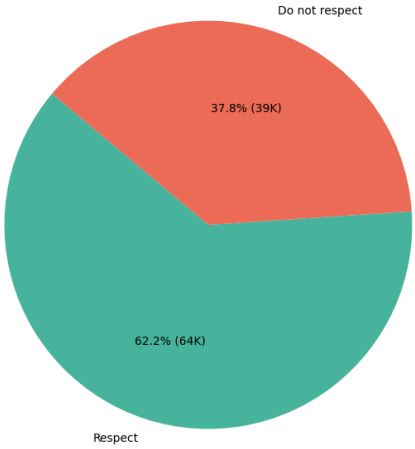


Figure 8.12: Diagram compliance with the Diagram Size principle.

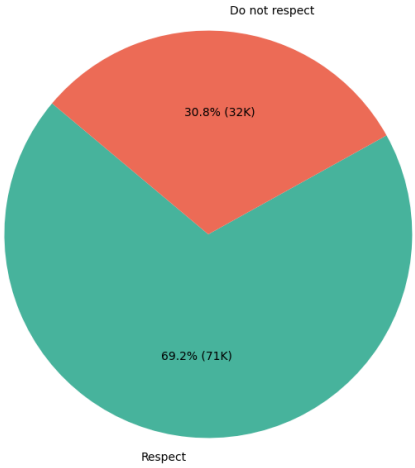


Figure 8.13: Diagram compliance with the Dual Coding principle.

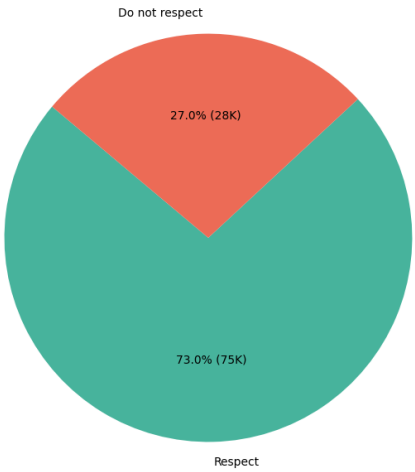


Figure 8.14: Diagram compliance with the Close Lines principle.

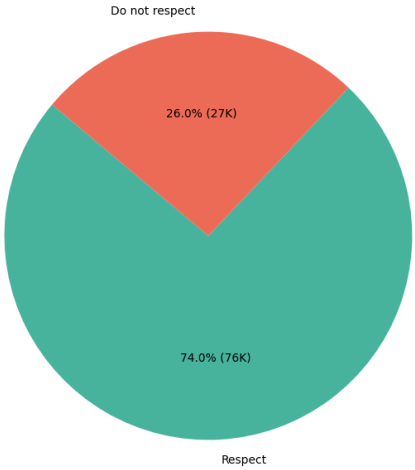


Figure 8.15: Diagram compliance with the Overlapping Elements principle.

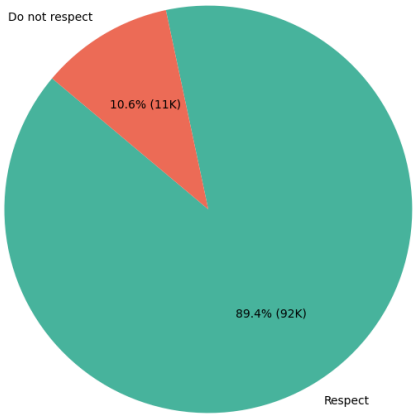


Figure 8.16: Diagram compliance with the Graphic Economy principle.

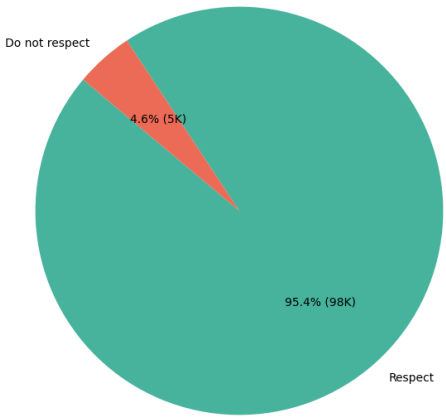


Figure 8.17: Diagram compliance with the Perceptual Discriminability principle.

Furthermore, we studied the distribution of the diagrams concerning the number of principle violations, for each principle. Specifically, this is important because it allows us to identify the predominant frequency of non-adherence to each principle within these diagrams.

The following analysis presents histograms, density plots and box plots. In particular, box plots show the spread of information, displaying the minimum, first quartile, median, third quartile and maximum. Additionally, it may show outliers. Figure 8.18 illustrates the main concepts of box plots.

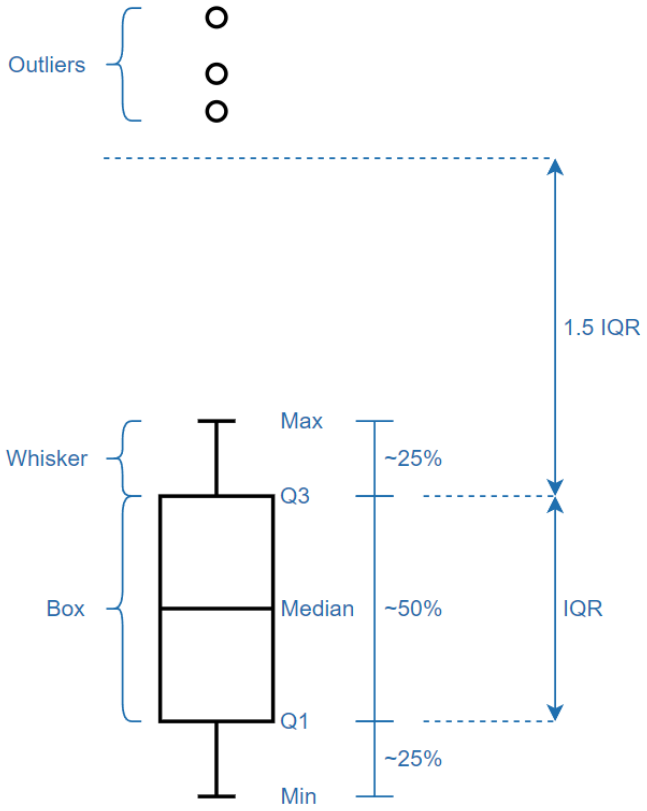


Figure 8.18: Main concepts of box plots.

Box plots have the following elements:

- **Box** - contains the values between the lower quartile Q1 and the upper quartile Q3, comprising approximately 50% of the data sample. The lower quartile Q1 and upper quartile Q3 correspond to the box limits. It also contains the median, which is the middle value of a data sample.
- **Whiskers** - go from each quartile to the minimum or maximum data points. These can extend up to 1.5 times the Interquartile Range (IQR), where the IQR represents the size of the box and can be calculated as the difference between the third quartile (Q3) and the first quartile (Q1). Furthermore, each whisker represents approximately 25% of the data sample.

- **Outliers** - any data point greater than 1.5 times the IQR below the first quartile (Q1) or above the third quartile (Q3).

The distribution of the diagrams was analysed based on the frequency of violations for the Visual Expressiveness principle, as shown in Figure 8.19. The analysis revealed that diagrams mostly have 1 to 41 occurrences of non-compliance with this principle, representing 93.33% of diagrams not adhering to this principle. Among these, the most frequent are diagrams containing 20 occurrences.

Also, the distribution of occurrences of non-compliance with the Visual Expressiveness principle was studied, as shown in Figure 8.20 (density plot) and Figure 8.21 (box plot). The analysis showed that the median number of occurrences is 20, and the average number of occurrences is 23. Moreover, the occurrences are spread in the range of 1 to 52 (8.21a), and there is a high number of outliers in the range of 52 to 500 (8.21b).

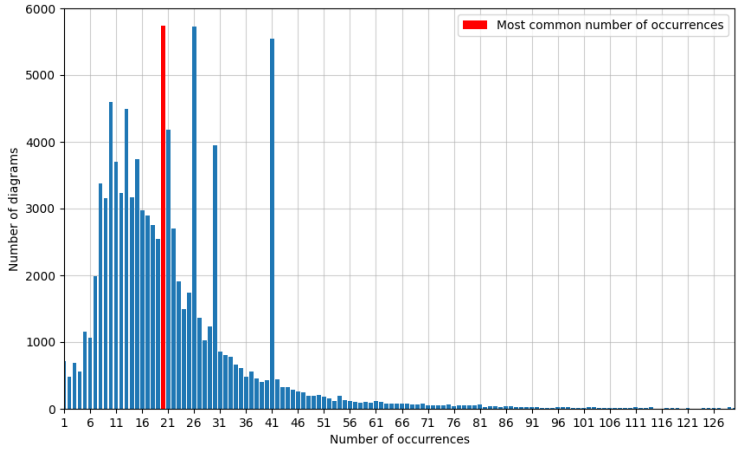


Figure 8.19: Distribution of diagrams based on the frequency of violations of the Visual Expressiveness principle (with axes limited for clarity).

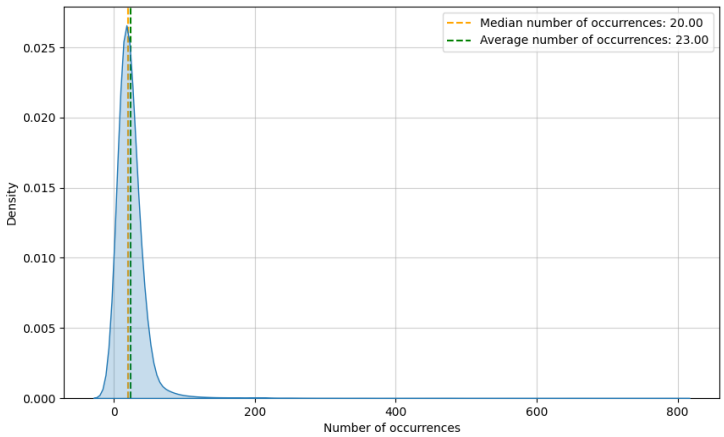
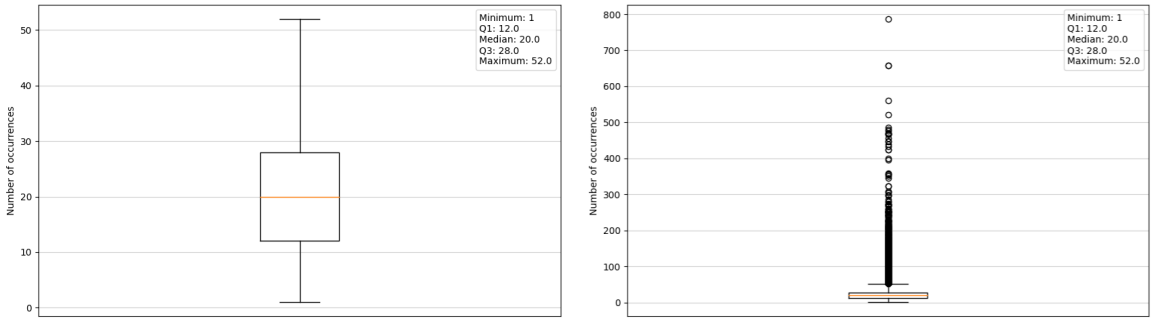


Figure 8.20: Density plot based on the frequency of violations of the Visual Expressiveness principle.



(a) Box plot without outliers.

(b) Box plot with outliers.

Figure 8.21: Box plot based on the frequency of violations of the Visual Expressiveness principle.

The distribution of the diagrams was analysed based on the frequency of violations for the Perceptual Discriminability principle, as shown in Figure 8.22. The analysis revealed that diagrams mainly have 1 to 13 occurrences of non-compliance with this principle, representing 87.42% of diagrams not adhering to this principle. Among these, the most frequent are diagrams containing one occurrence.

Furthermore, the distribution of occurrences of non-compliance with the Perceptual Discriminability principle was studied, as shown in Figure 8.23 (density plot) and Figure 8.24 (box plot). The analysis showed that the median number of occurrences is 2, and the average number of occurrences is 15.32. Further, the occurrences are spread in the range of 1 to 13.5 (8.24a), and there is a high number of outliers in the range of 13.5 to 600 (8.24b).

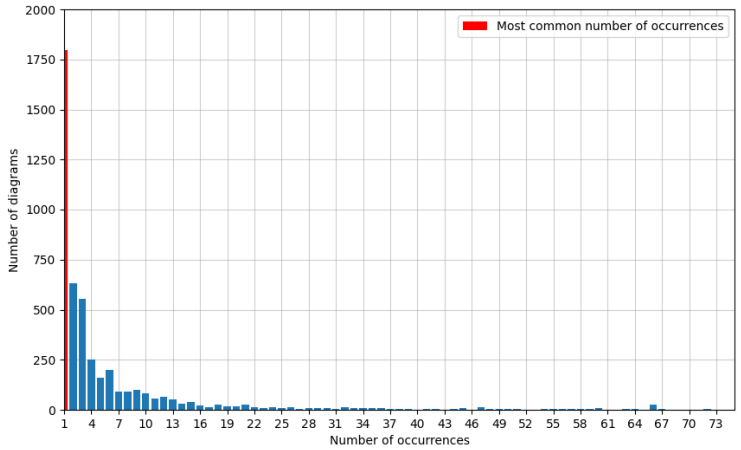


Figure 8.22: Distribution of diagrams based on the frequency of violations of the Perceptual Discriminability principle (with axes limited for clarity).

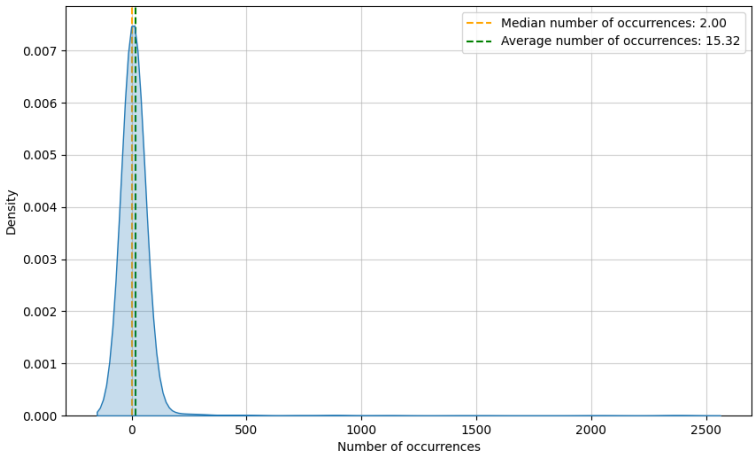
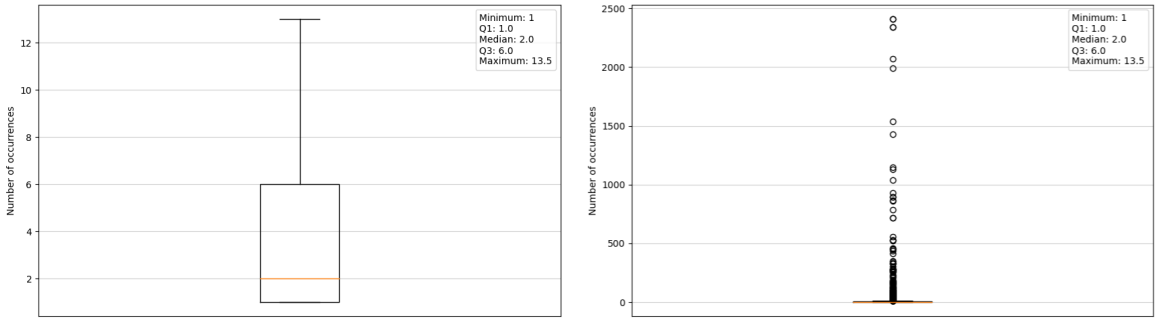


Figure 8.23: Density plot based on the frequency of violations of the Perceptual Discriminability principle.



(a) Box plot without outliers.

(b) Box plot with outliers.

Figure 8.24: Box plot based on the frequency of violations of the Perceptual Discriminability principle.

The distribution of the diagrams was analysed based on the frequency of violations for the Overlapping Elements principle, as shown in Figure 8.25. The analysis revealed that diagrams mainly have 1 to 20 occurrences of non-compliance with this principle, representing 85.23% of diagrams not adhering to this principle. Within this range, the most frequent are diagrams containing 20 occurrences.

Furthermore, the distribution of occurrences of non-compliance with the Overlapping Elements principle was studied, as shown in Figure 8.26 (density plot) and Figure 8.27 (box plot). The analysis showed that the median number of occurrences is 10, and the average number of occurrences is 22.88. Additionally, the occurrences are spread in the range of 1 to 42.5 (8.27a), and there is a high number of outliers in the range of 42.5 to 2,000 (8.27b).

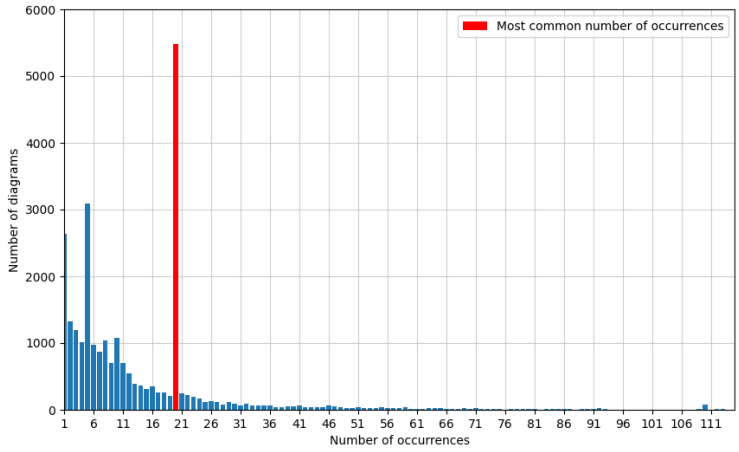


Figure 8.25: Distribution of diagrams based on the frequency of violations of the Overlapping Elements principle (with axes limited for clarity).

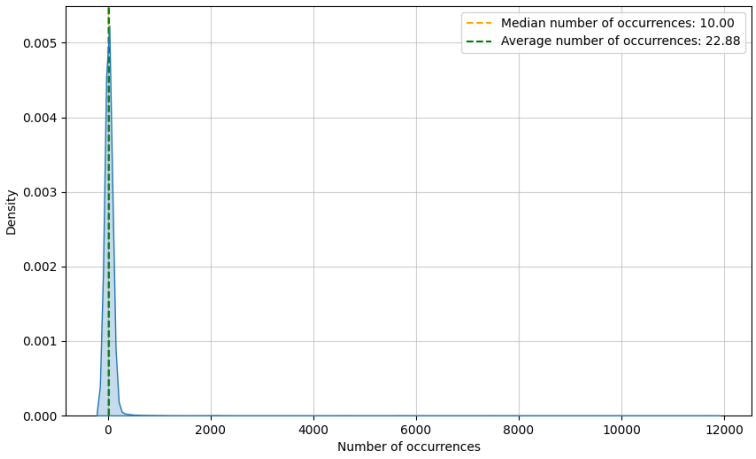
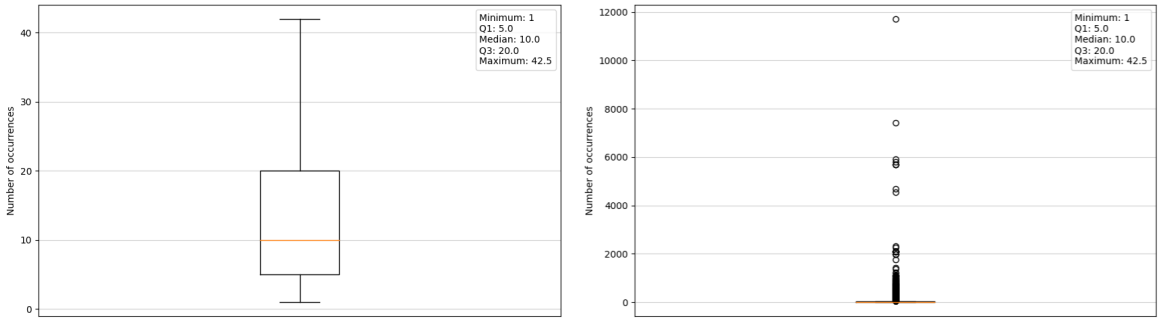


Figure 8.26: Density plot based on the frequency of violations of the Overlapping Elements principle.



(a) Box plot without outliers.

(b) Box plot with outliers.

Figure 8.27: Box plot based on the frequency of violations of the Overlapping Elements principle.

The distribution of the diagrams was analysed based on the frequency of violations for the Intersections principle, as shown in Figure 8.28. The analysis revealed that diagrams mostly have 1 to 30 occurrences of non-compliance with this principle, representing 87.49% of diagrams not adhering to this principle. Within this set, the most frequent are diagrams containing one occurrence.

Also, the distribution of occurrences of non-compliance with the Intersections principle was studied, as shown in Figure 8.29 (density plot) and Figure 8.30 (box plot). The analysis revealed that the median number of occurrences is 5, and the average number of occurrences is 30.04. Additionally, the occurrences are spread in the range of 1 to 29.5 (8.30a), and there is a high number of outliers in the range of 29.5 to 5,000 (8.30b).

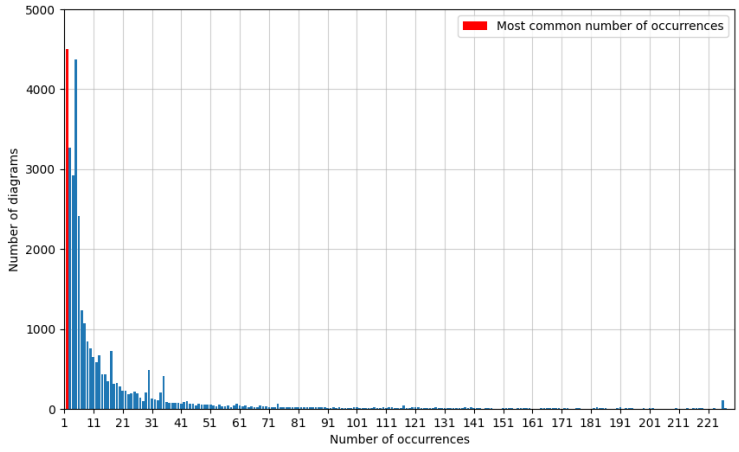


Figure 8.28: Distribution of diagrams based on the frequency of violations of the Intersections principle (with axes limited for clarity).

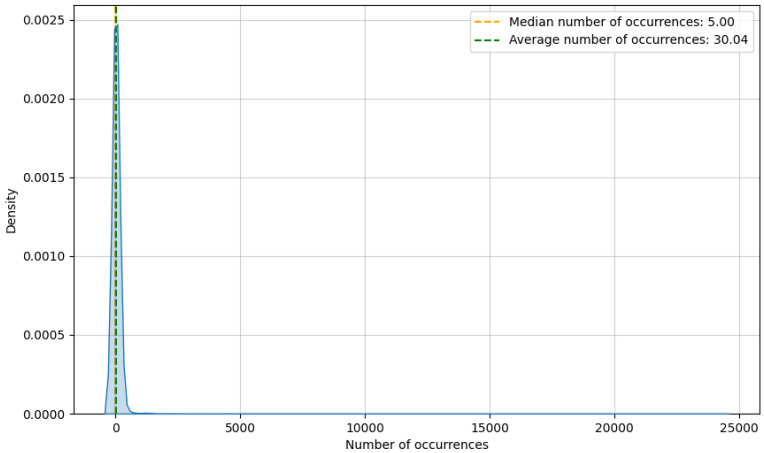
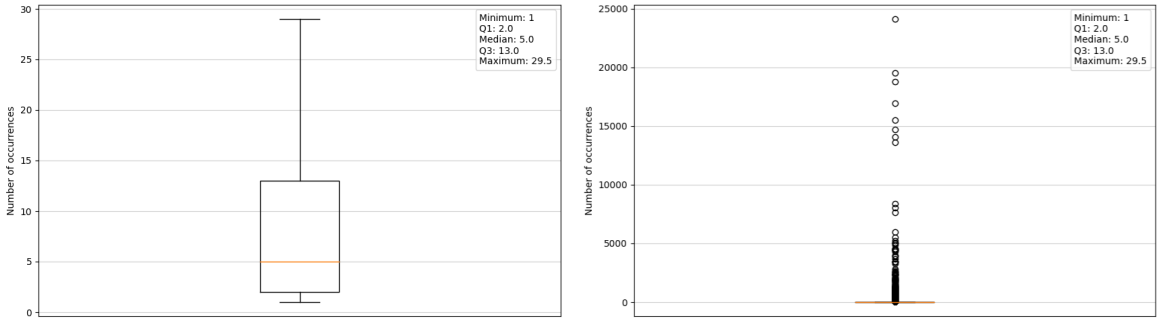


Figure 8.29: Density plot based on the frequency of violations of the Intersections principle.



(a) Box plot without outliers.

(b) Box plot with outliers.

Figure 8.30: Box plot based on the frequency of violations of the Intersections principle.

The distribution of the diagrams was analysed based on the frequency of violations for the Dual Coding principle, as shown in Figure 8.31. The analysis revealed that diagrams mainly have 1 to 11 occurrences of non-compliance with this principle, representing 92.51% of diagrams not adhering to this principle. Among these, the most frequent are diagrams containing one occurrence.

Also, the distribution of occurrences of non-compliance with the Dual Coding principle was studied, as shown in Figure 8.32 (density plot) and Figure 8.33 (box plot). The analysis showed that the median number of occurrences is 2, and the average number of occurrences is 4.42. Additionally, the occurrences are spread in the range of 1 to 11 (8.33a), and there is a high number of outliers in the range of 11 to 200 (8.33b).

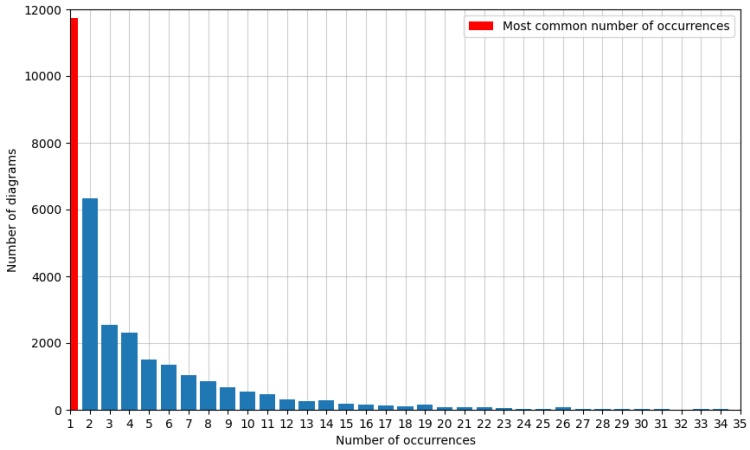


Figure 8.31: Distribution of diagrams based on the frequency of violations of the Dual Coding principle (with axes limited for clarity).

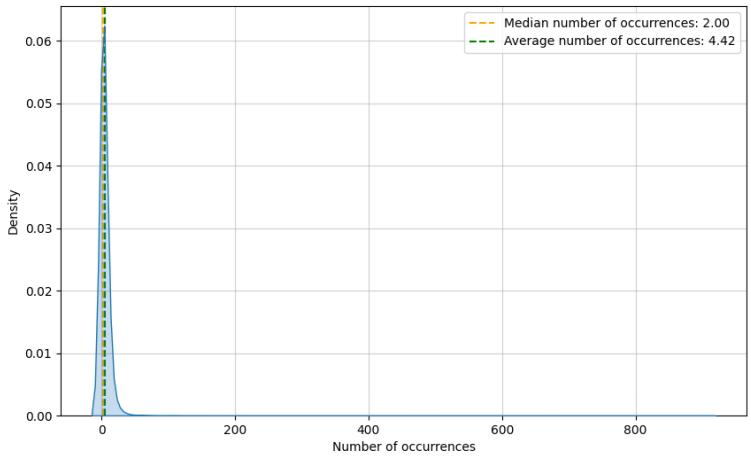
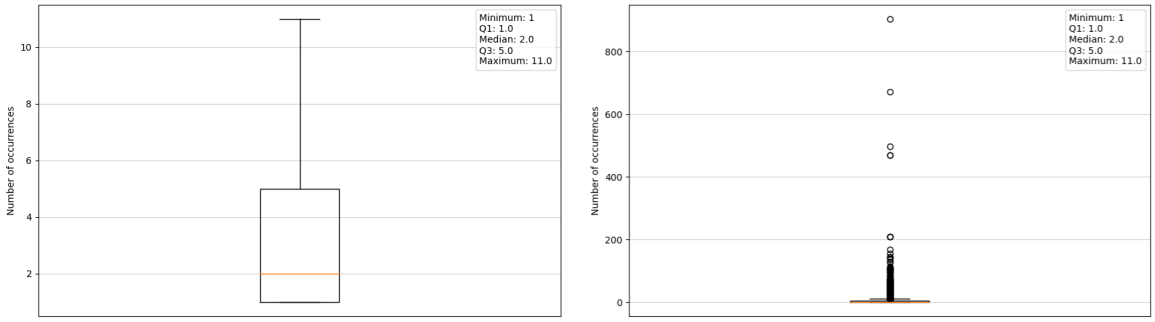


Figure 8.32: Density plot based on the frequency of violations of the Dual Coding principle.



(a) Box plot without outliers.

(b) Box plot with outliers.

Figure 8.33: Box plot based on the frequency of violations of the Dual Coding principle.

The distribution of the diagrams was analysed based on the frequency of violations for the Close Lines principle, as shown in Figure 8.34. The analysis revealed that diagrams mostly have 1 to 60 occurrences of non-compliance with this principle, representing 86.33% of diagrams not adhering to this principle. Among these, the most frequent are diagrams containing ten occurrences.

Furthermore, the distribution of occurrences of non-compliance with the Close Lines principle was studied, as shown in Figure 8.35 (density plot) and Figure 8.36 (box plot). The analysis showed that the median number of occurrences is 10, and the average number of occurrences is 79.80. Additionally, the occurrences are spread in the range of 1 to 74 (8.36a), and there is a high number of outliers in the range of 74 to 10,000 (8.36b).

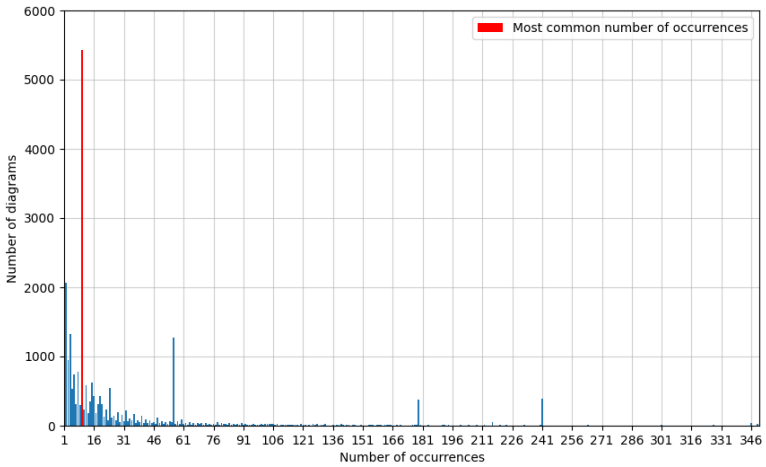


Figure 8.34: Distribution of diagrams based on the frequency of violations of the Close Lines principle (with axes limited for clarity).

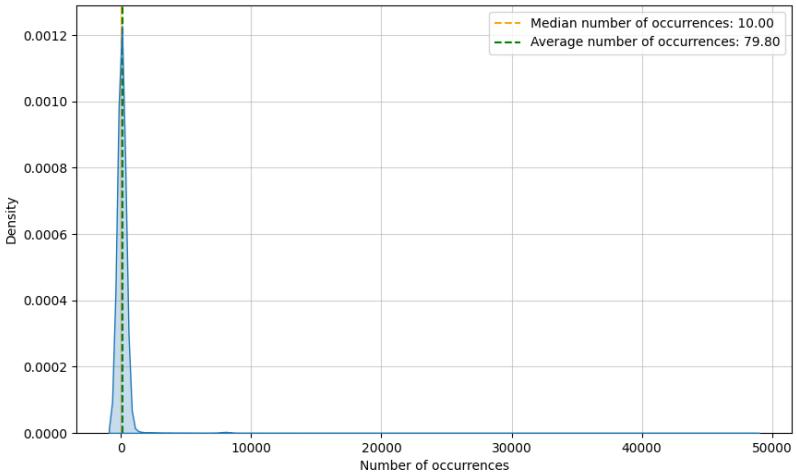
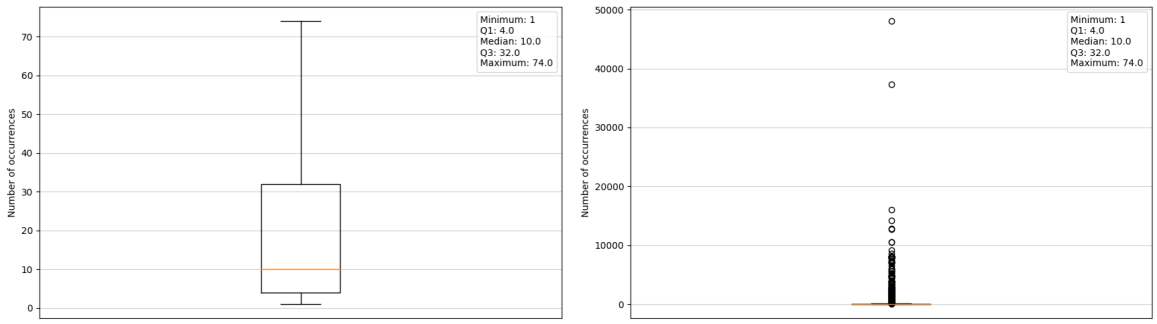


Figure 8.35: Density plot based on the frequency of violations of the Close Lines principle.



(a) Box plot without outliers.

(b) Box plot with outliers.

Figure 8.36: Box plot based on the frequency of violations of the Close Lines principle.

The distribution of the diagrams was analysed based on the frequency of violations for the Bends in Lines principle, as shown in Figure 8.37. The analysis revealed that diagrams mostly have 1 to 18 occurrences of non-compliance with this principle, representing 94.26% of diagrams not adhering to this principle. Within this, the most frequent are diagrams containing four occurrences.

Furthermore, the distribution of occurrences of non-compliance with the Close Lines principle was studied, as shown in Figure 8.38 (density plot) and Figure 8.39 (box plot). The analysis revealed that the median number of occurrences is 5, and the average number of occurrences is 7.64. Additionally, the occurrences are spread in the range of 1 to 14 (8.39a), and there is a high number of outliers in the range of 14 to 300 (8.39b).

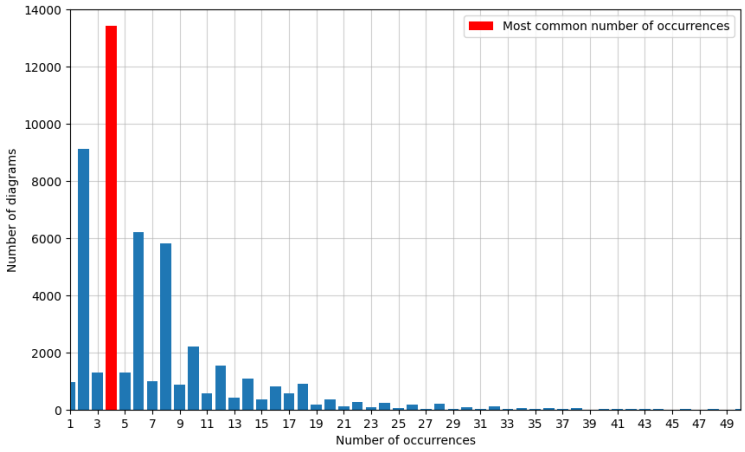


Figure 8.37: Distribution of diagrams based on the frequency of violations of the Bends in Lines principle (with axes limited for clarity).

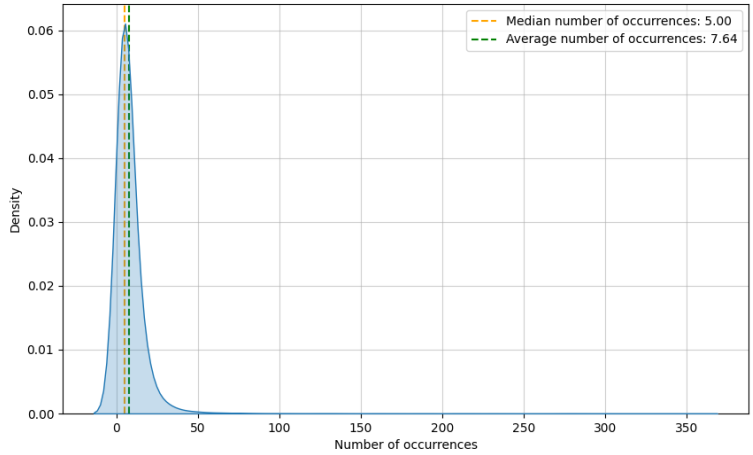
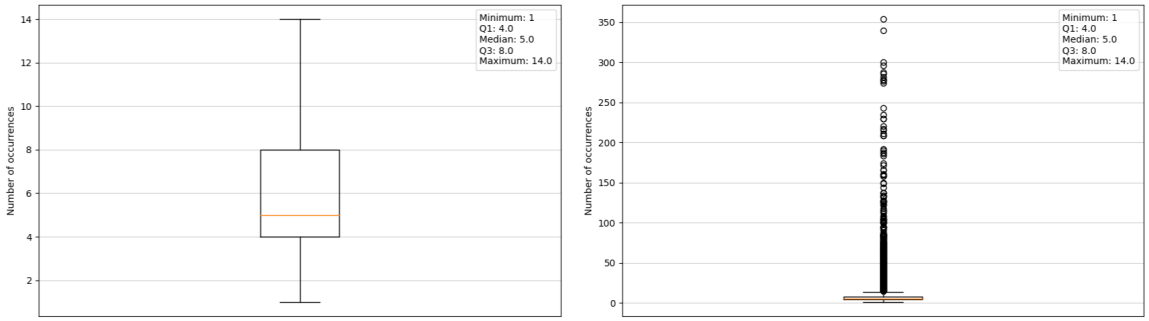


Figure 8.38: Density plot based on the frequency of violations of the Bends in Lines principle.



(a) Box plot without outliers.

(b) Box plot with outliers.

Figure 8.39: Box plot based on the frequency of violations of the Bends in Lines principle.

Note that this analysis did not consider the principles of Diagram Size and Graphic Economy, as each diagram can only violate each of these principles once.

Table 8.2 summarizes the above analysis.

Table 8.2: Summary of the distribution of occurrences of non-adherence to each principle.

Principle	**Occurrences of non-adherence**					
	Min	**Q1**	**Median**	**Mean**	**Q3**	**Max**
Visual Expressiveness	1	12	20	23	28	52
Perceptual Discriminability	1	1	2	15.32	6	13.5
Overlapping Elements	1	5	10	22.88	20	42.5
Intersections	1	2	5	30.04	13	29.5
Dual Coding	1	1	2	4.42	5	11
Close Lines	1	4	10	79.80	32	74
Bends in Lines	1	4	5	7.64	8	14

Subsequently, regarding to the Graphic Economy principle, the distribution of diagrams was studied based on the number of different element types, as illustrated in Figure 8.40 (histogram) and Figure 8.41 (density plot). Addressing this concern was essential, as we only analysed whether or not the diagrams respected this principle, and it is also important to understand the distribution of the number of different element types. Figure 8.40 shows in green the bins where the number of different element types is less than or equal to six (graphical complexity threshold), and in red the bins where the number of different element types surpasses six.

This showed that 89.4% of the dataset adheres to this principle, while only 10.6% does not. Furthermore, we found that three different element types are the most frequently used in the diagrams of the dataset. Also, diagrams that do not follow this principle present between 7 and 12 different element types, with seven being the most common. Moreover, none of the diagrams in the dataset contain the entire graphical vocabulary of

UML class diagrams, which is sixteen.

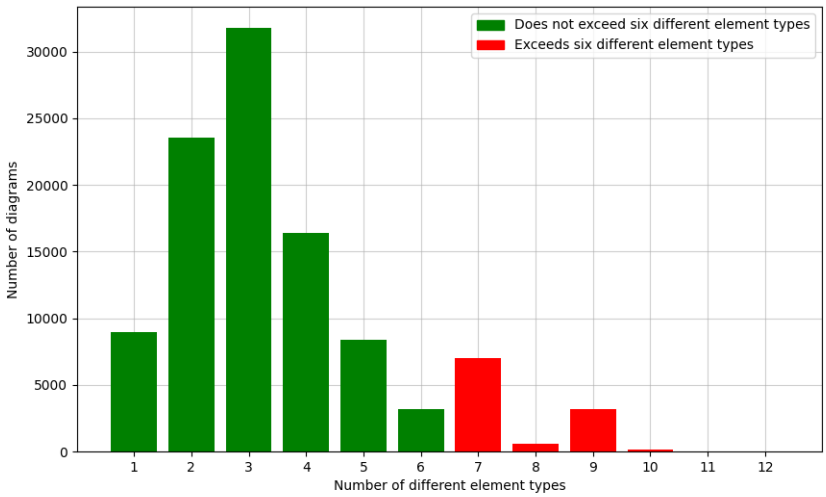


Figure 8.40: Distribution of diagrams based on the number of different element types (Graphic Economy principle).

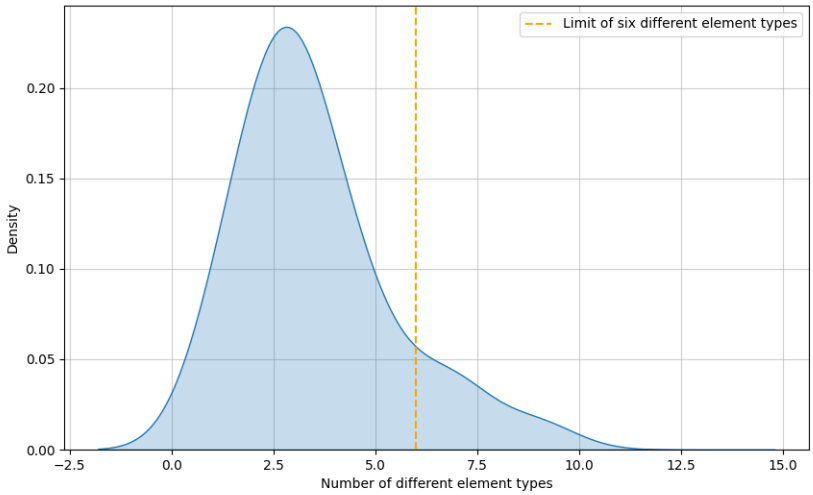


Figure 8.41: Density plot based on the number of different element types (Graphic Economy principle).

Subsequently, concerning the Diagram Size principle, the distribution of diagrams was studied based on the number of elements they encompass, as illustrated in Figure 8.42 (histogram) and Figure 8.43 (density plot). Specifically, this is important since we only analysed whether or not the diagrams respected this principle, and it is also crucial to comprehend the distribution of the number of elements and whether they are in the desired interval of 20 to 60 elements.

This revealed that 62.2% of the dataset adheres to this principle, while 37.8% does not. Furthermore, outside the ideal range, the range of 1 to 19 elements presents a higher

frequency of diagrams than the range of 61 to 1907. Moreover, the further the number of elements is from the ideal range, the less ideal it becomes. This is evident in the interval from 61 to 1907, which extends until 1907. As the distribution shows, in this interval there is a significant frequency of diagrams between 61 and 250, accounting for 9,819 diagrams (9.52% of the dataset). However, between 250 and 1907, there are only 383 diagrams (0.37% of the dataset).

Also, outside the ideal range, ten elements are the most common in diagrams, while in the ideal range is 21 elements.

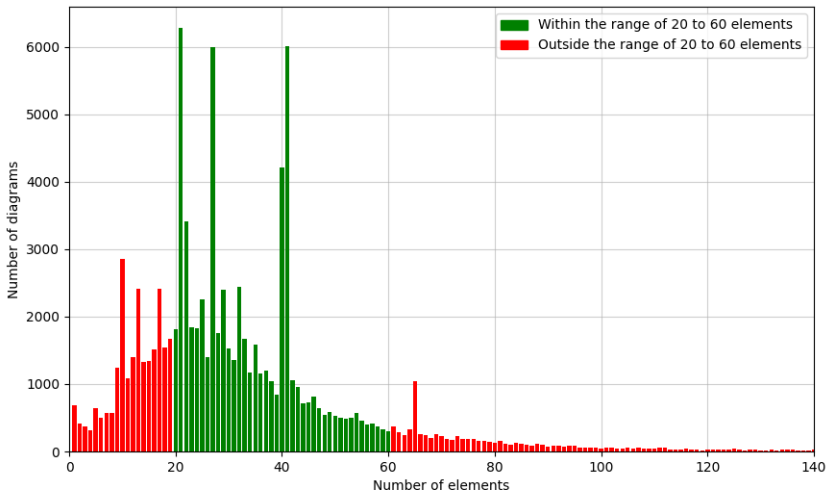


Figure 8.42: Distribution of diagrams based on the number of elements (with axes limited for clarity) (Diagram Size principle).

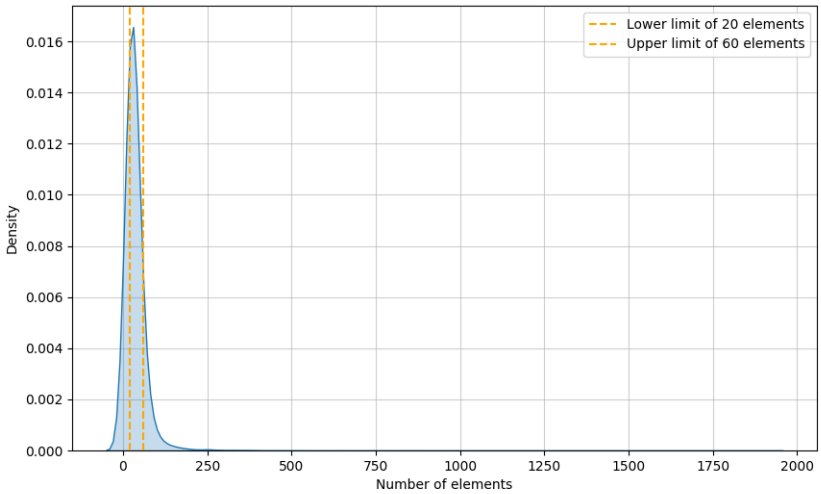


Figure 8.43: Density plot based on the number of elements (Diagram Size principle).

8.3 Discussion of Results

The main objective of this dissertation is to identify the most frequent defects present in UML class diagrams extracted from public repositories. The previous section shows that we achieved this objective. In addition to the results presented in the previous sections, it is also necessary to discuss them.

Below, we discuss some considerations arising from the analysis of the results:

- Regarding the general results of the analysis of the dataset (section 8.1), the characteristics of a typical UML class diagram from the dataset are: 1 to 26 entities, 1 to 22 relationships, the most used entities are classes, and the most used relationships are associations and generalizations. These findings seem natural to us about what a "normal" UML class diagram looks like. The prevalence of classes as the most frequently used entities is consistent with the fundamental role that classes play in UML class diagrams. Similarly, the prominence of associations and generalizations as the most used relationships aligns with the core principles of object-oriented systems. Associations represent the connections and interactions between entities, while generalizations define inheritance hierarchies.

- Regarding the analysis of the results obtained from the evaluation tool (section 8.2), the most common defects present in the collected UML class diagrams (total number of occurrences) are the violation of the Close Lines principle (with 2,221,340 occurrences) and the Visual Expressiveness principle (with 2,216,424 occurrences). We expected that the Close Lines principle would not be respected to a large extent, as it typically poses a considerable challenge to maintain the aesthetics of a diagram. The high number of occurrences suggests that diagram modellers may have had difficulty keeping the relationship lines distant from each other. On the other hand, we were surprised that the Visual Expressiveness principle was the second principle with the most occurrences of non-adherence. With this principle, we check whether the graphical elements of the diagrams employ colours beyond their default colour. Note that the process of colouring an element is very quick and intuitive to perform in the GenMyModel editor, from an interaction standpoint. Therefore, this can be ruled out as a cause of the infrequent use of colours. A possible explanation could be that many of the diagrams are relatively small, often containing fewer than 26 entities. In such cases, the need for extensive colour coding might be less pronounced, as the simplicity and size of the diagram may not necessitate the use of color.

- Continuing our examination of the total number of occurrences of each principle not respected, we were also surprised by the high number of occurrences of non-adherence to the Intersections principle (with 1,287,512 occurrences) and the Overlapping Elements principle (with 612,577 occurrences). Concerning the Intersections principle, the results suggest that there may have been challenges in organizing

the relationships in the diagram to reduce intersections. As for the Overlapping Elements principle, a possible explanation is that diagrams with a high number of elements can contribute to the overlapping of elements. However, it is important to note that the GenMyModel editor automatically resizes its canvas when elements are added to the canvas borders. So, while diagram size may be a factor, the resizing feature should theoretically mitigate this issue by expanding beyond the canvas boundaries.

- Still, regarding the total number of occurrences of each principle not respected, we expected that the Diagram Size and Graphic Economy principles would be those with the lowest number of occurrences of non-adherence (as confirmed by the results), as for these principles there can only be one occurrence per diagram. Concerning the Perceptual Discriminabilty principle, we also anticipated that there would be a low number of occurrences, which was demonstrated by the results. As typically, the elements used can be differentiated from each other, due to the use of different types of elements or the same type of elements, but with different names, attributes, operations, enumeration literals (if applicable), and (as already seen, to a lesser extent) colours. As for the Dual coding principle, we found that occurrences of non-adherence are relatively low. This result aligns with our expectations, since when a relationship is created the GenMyModel editor automatically inserts a default name and multiplicity (if applicable) into the relationship.

- Regarding the number of diagrams in which each principle is not respected at least once, we found that the Visual Expressiveness, Bends in Lines and Intersections principles, are the principles that the diagrams of the dataset do not comply with the most, with 93.5%, 50% and 41.6% of non-adherence, respectively. The percentage for the Visual Expressiveness principle corroborates the results of the total number of occurrences of non-adherence, being the second principle with more occurrences. As for Bends in Lines principles, we expected the percentage of non-adherence to be high or even higher (than 50%), as diagrams normally have relationships with bends in their lines. Concerning the Intersections principle, the percentage is in line with the results of the total number of occurrences of non-adherence, being the third principle with more occurrences.

- Continuing our analysis of the number of diagrams in which each principle is not respected at least once, we found that only 27% of the diagrams in the dataset do not adhere to the Close Lines principle. We expected this percentage to be higher, as this principle is the one that presents the highest total number of non-adherence occurrences. This indicates that a small percentage of diagrams in the dataset contain a high number of cases of non-compliance with this principle. Similarly, we found that only 26% of the diagrams in the dataset do not adhere to the Overlapping Elements principle. However, this principle is the fourth with the total number of

non-adherence occurrences. Therefore, this also indicates that a small percentage of diagrams in the dataset contain a high number of cases of non-compliance with this principle.

9

Conclusions

This chapter presents the conclusions of this dissertation. Then, it outlines its contributions. Finally, it describes the future work.

9.1 Conclusions

This dissertation had three main objectives. First, create a web scraping tool to collect UML class diagrams from online repositories. Second, create an automated evaluation tool to analyse and evaluate the collected dataset of UML class diagrams to identify the defects present in each diagram in the dataset. Third, to analyse the results obtained from the automated evaluation tool and to identify the most common defects present in these diagrams.

We started by collecting the UML class diagrams that would later be used to conduct the evaluation. For this, a web scraping tool was created to browse the projects on the GenMyModel platform and download the XMI files of the projects that contained a UML class diagram. The outcome was the successful creation of a dataset encompassing 103,103 XMI files, all containing a UML class diagram.

After creating the dataset, we evaluated the collected dataset of more than one hundred thousand diagrams to identify the defects present in these diagrams. To this end, an automated evaluation tool was developed to individually analyse each diagram in the dataset and identify the defects present in them. This tool employed the principles of the PoN and the principles of diagram size and diagram flaws from [Stö16] to identify the defects.

By analysing the results obtained from running the evaluation tool, we were able to identify the most common defects present in the collected UML class diagrams.

9.2 Contributions

The first contribution of this dissertation is the development of a web scraping tool to collect UML class diagrams from the GenMyModel platform. This web scraper browses

each public project within the GenMyModel platform repository and downloads the XMI files of projects containing a UML class diagram.

The second contribution is the collection of a dataset consisting of 103,103 XMI files, all containing a UML class diagram. This dataset was created by running the web scraping tool in the Google Colab platform. Additionally, this dataset is made publicly available to address the scarcity of datasets with this type of diagram.

The third contribution is the development of an automated evaluation tool designed to evaluate the collected dataset and identify the defects present in each model. This tool employs principles of the PoN, as well as the principles of diagram size and diagram flaws from [Stö16] to identify these defects.

Finally, the last contribution is the analysis of the results obtained from the execution of the automated evaluation tool, which identifies the most common defects present in the collected models.

9.3 Future Work

For future work, regarding the web scraping tool, it currently only collects UML class diagrams from the GenMyModel platform. However, this platform hosts other types of UML diagrams, such as use cases, activity and sequence diagrams. Furthermore, apart from UML, it also supports other modelling languages, including BPMN and Archimate. Thus, this tool can be extended to collect other types of diagrams.

As for the automated evaluation tool, it identifies the defects present in the UML class diagrams using the principles of PoN and the principles of diagram size and diagram flaws from [Stö16]. However, this tool can be extended to include other principles to identify other types of defects.

Still regarding the automated evaluation tool, one aspect to take into account is the analysis of the evaluation results obtained from running this tool. Although these results can be analysed automatically through a second Python application, this process can be improved by merging the two applications, so that once the automated evaluation tool has finished evaluating the dataset, it can also create the analysis of the results.

Finally, the present study has focused on evaluating a dataset of UML class diagrams to find the most common defects present in these models. However, this study can be replicated to find the most common defects present in other types of UML diagrams, such as use case, component and deployment diagrams.

Bibliography

[AS09] B. de Alwis and J. Sillito. "Why are software projects moving from centralized to decentralized version control systems?" In: *2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering*. 2009, pp. 36–39 (cit. on pp. 6, 7).

[Apa] Apache. *Apache Subversion*. URL: https://subversion.apache.org/ (visited on 2022-11-04) (cit. on p. 7).

[Ath] A. Athalyen. *Version Control (Git)*. URL: https://missing.csail.mit.edu/2020/version-control/ (visited on 2022-11-25) (cit. on pp. 6, 8, 9).

[Axe] Axellience. *GenMyModel*. URL: https://www.genmymodel.com/ (visited on 2023-09-17) (cit. on pp. 26, 32).

[Azu] M. Azure. *Azure DevOps Server*. URL: https://azure.microsoft.com/en-us/products/devops/server (visited on 2023-01-11) (cit. on p. 7).

[BH00] F. Barbier and B. Henderson-Sellers. "Object modelling languages: An evaluation and some key expectations for the future". In: *Annals of Software Engineering* 10.1-4 (2000), pp. 67–101 (cit. on p. 2).

[Ber83] J. Bertin. *Semiology of graphics*. University of Wisconsin press, 1983 (cit. on pp. 16, 21).

[Bir+09] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu. "The promises and perils of mining git". In: *2009 6th IEEE International Working Conference on Mining Software Repositories*. 2009, pp. 1–10 (cit. on p. 7).

[BD00] J. Blankenship and D. F. Dansereau. "The effect of animated node-link displays on information recall". In: *The Journal of Experimental Education* 68.4 (2000), pp. 293–308 (cit. on p. 23).

[BB16] E. J. Braude and M. E. Bernstein. *Software engineering: modern approaches*. Waveland Press, 2016 (cit. on p. 1).

[BJ99] C. Britton and S. Jones. "The untrained eye: how languages for software specification support understanding in untrained users". In: *Human–Computer Interaction* 14.1-2 (1999), pp. 191–244 (cit. on pp. 18, 23).

[CS14] S. Chacon and B. Straub. *Pro Git*. 2nd. USA: Apress, 2014. ISBN: 1484200772 (cit. on pp. 8, 9).

[cur] curl. *curl*. URL: https://curl.se/ (visited on 2023-04-26) (cit. on p. 14).

[Due+18] S. Dueñas, V. Cosentino, G. Robles, and J. M. Gonzalez-Barahona. "Perceval: Software Project Data at Your Will". In: *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. 2018, pp. 1–4 (cit. on p. 12).

[El +15] A. El Kouhen, A. Gherbi, C. Dumoulin, and F. Khendek. "On the Semantic Transparency of Visual Notations: Experiments with UML". In: *SDL 2015: Model-Driven Engineering for Smart Cities*. Cham: Springer International Publishing, 2015, pp. 122–137 (cit. on p. 28).

[Ele] ElementTree. *The ElementTree XML API*. URL: https://docs.python.org/3/library/xml.etree.elementtree.html (visited on 2023-05-11) (cit. on p. 53).

[FC93] R. L. Flood and E. Carson. *Dealing with complexity: an introduction to the theory and application of systems science*. Springer Science & Business Media, 1993 (cit. on p. 20).

[Fou] T. L. Foundation. *PyTorch*. URL: https://pytorch.org/ (visited on 2023-08-13) (cit. on p. 35).

[Gee] Geekflare. *Google Colab: Everything you Need to Know*. URL: https://geekflare.com/google-colab/ (visited on 2023-08-13) (cit. on p. 35).

[Gita] Git. *Git*. URL: https://git-scm.com/ (visited on 2022-11-05) (cit. on p. 7).

[Gitb] GitHub. *GitHub*. URL: https://github.com/ (visited on 2022-11-18) (cit. on pp. 2, 10).

[Gitc] GitHub. *GitHub REST API*. URL: https://docs.github.com/en/rest (visited on 2022-11-18) (cit. on p. 10).

[Gitd] GitLab. *GitLab*. URL: https://about.gitlab.com/ (visited on 2022-11-22) (cit. on p. 10).

[Gite] GitLab. *The benefits of a distributed version control system*. URL: https://about.gitlab.com/topics/version-control/benefits-distributed-version-control-system/ (visited on 2023-01-11) (cit. on p. 7).

[Gitf] GitLab. *What is a centralized version control system*. URL: https://about.gitlab.com/topics/version-control/what-is-centralized-version-control-system/ (visited on 2023-01-11) (cit. on p. 7).

[Gitg] GitLab. *What is version control?* URL: https://about.gitlab.com/topics/version-control/ (visited on 2023-01-31) (cit. on p. 6).

[Goo] Google. *Google BigQuery*. URL: https://cloud.google.com/bigquery (visited on 2022-11-18) (cit. on p. 10).

[Gos+21] B. Gosala, S. R. Chowdhuri, J. Singh, M. Gupta, and A. Mishra. "Automatic Classification of UML Class Diagrams Using Deep Learning Technique: Convolutional Neural Network". In: *Applied Sciences* 11.9 (2021). ISSN: 2076-3417 (cit. on p. 12).

[Gou13] G. Gousios. "The GHTorent Dataset and Tool Suite". In: *Proceedings of the 10th Working Conference on Mining Software Repositories*. MSR '13. San Francisco, CA, USA: IEEE Press, 2013, pp. 233–236. ISBN: 9781467329361 (cit. on p. 10).

[GS12] G. Gousios and D. Spinellis. "GHTorrent: Github's data from a firehose". In: *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. 2012, pp. 12–21 (cit. on p. 10).

[Gou+14] G. Gousios, B. Vasilescu, A. Serebrenik, and A. Zaidman. "Lean GHTorrent: GitHub Data on Demand". In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014. Association for Computing Machinery, 2014, pp. 384–387. ISBN: 9781450328630 (cit. on p. 11).

[Has08] A. E. Hassan. "The road ahead for Mining Software Repositories". In: *2008 Frontiers of Software Maintenance*. 2008, pp. 48–57 (cit. on p. 10).

[Heb+16] R. Hebig, T. H. Quang, M. R. V. Chaudron, G. Robles, and M. A. Fernandez. "The Quest for Open Source Projects That Use UML: Mining GitHub". In: *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. MODELS '16. Saint-malo, France: Association for Computing Machinery, 2016, pp. 173–183. ISBN: 9781450343213 (cit. on pp. 11, 12, 25, 26).

[HSA20] T. S. Heinze, V. Stefanko, and W. Amme. "Mining BPMN Processes on GitHub for Tool Validation and Development". In: *Enterprise, Business-Process and Information Systems Modeling*. Springer, 2020, pp. 193–208. ISBN: 978-3-030-49418-6 (cit. on pp. 2, 10, 11, 13, 26).

[Jup] P. Jupyter. *Jupyter Notebook*. URL: https://jupyter.org/ (visited on 2023-08-13) (cit. on p. 34).

[KCM07] H. Kagdi, M. L. Collard, and J. I. Maletic. "A survey and taxonomy of approaches for mining software repositories in the context of software evolution". In: *Journal of software maintenance and evolution: Research and practice* 19.2 (2007), pp. 77–131 (cit. on pp. 2, 10).

[KC13] B. Karasneh and M. R. Chaudron. "Img2UML: A System for Extracting UML Models from Images". In: *2013 39th Euromicro Conference on Software Engineering and Advanced Applications*. 2013, pp. 134–137 (cit. on pp. 12, 31).

[Ker] Keras. *Keras*. URL: https://keras.io/ (visited on 2023-08-13) (cit. on p. 35).

[Khd21] M. Khder. "Web Scraping or Web Crawling: State of Art, Techniques, Approaches and Application". In: *International Journal of Advances in Soft Computing and its Applications* 13 (2021-12), pp. 145–168 (cit. on pp. 2, 13–15).

[KHH00] J. Kim, J. Hahn, and H. Hahn. "How do we understand a system with (so) many diagrams? Cognitive integration processes in diagrammatic reasoning". In: *Information Systems Research* 11.3 (2000), pp. 284–303 (cit. on p. 20).

[KA90] K. R. Koedinger and J. R. Anderson. "Abstract planning and perceptual chunks: Elements of expertise in geometry". In: *Cognitive Science* 14.4 (1990), pp. 511–550 (cit. on p. 23).

[Küh06] T. Kühne. "Matters of (meta-) modeling". In: *Software & Systems Modeling* 5.4 (2006), pp. 369–385 (cit. on pp. 1, 5, 6).

[LR96] J. Lamping and R. Rao. "The hyperbolic browser: A focus+ context technique for visualizing large hierarchies". In: *Journal of Visual Languages & Computing* 7.1 (1996), pp. 33–55 (cit. on p. 20).

[Loh93] G. L. Lohse. "A cognitive model for understanding graphical perception". In: *Human-Computer Interaction* 8.4 (1993), pp. 353–388 (cit. on p. 21).

[Lou21] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (cit. on p. ii).

[Luc] Lucidchart. *Types of UML Diagrams*. URL: https://www.lucidchart.com/blog/types-of-UML-diagrams (visited on 2023-09-24) (cit. on p. 1).

[Mac86] J. Mackinlay. "Automating the design of graphical presentations of relational information". In: *Acm Transactions On Graphics (Tog)* 5.2 (1986), pp. 110–141 (cit. on p. 21).

[Mat] Matplotlib. *Matplotlib*. URL: https://matplotlib.org/ (visited on 2023-08-13) (cit. on p. 35).

[MM03] R. E. Mayer and R. Moreno. "Nine ways to reduce cognitive load in multimedia learning". In: *Educational psychologist* 38.1 (2003), pp. 43–52 (cit. on p. 20).

[MZZ06] H. Mei, W. Zhang, and H. Zhao. "A metamodel for modeling system features and their refinement, constraint and interaction relationships". In: *Software & Systems Modeling* 5.2 (2006), pp. 172–186 (cit. on p. 6).

[Mil56] G. A. Miller. "The magical number seven, plus or minus two: Some limits on our capacity for processing information." In: *Psychological review* 63.2 (1956), p. 81 (cit. on pp. 19, 22, 45).

[Moo09] D. Moody. "The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering". In: *IEEE Transactions on Software Engineering* 35.6 (2009), pp. 756–779 (cit. on pp. 2, 14–22, 28, 43–46).

[MH09] D. Moody and J. van Hillegersberg. "Evaluating the visual syntax of UML: An analysis of the cognitive effectiveness of the UML family of diagrams". In: *Software Language Engineering: First International Conference, SLE 2008, Toulouse, France, September 29-30, 2008. Revised Selected Papers 1*. Springer. 2009, pp. 16–34 (cit. on p. 28).

[Mor+16] V. Moreno, G. Génova, M. Alejandres, and A. Fraga. "Automatic Classification of Web Images as UML Diagrams". In: *Proceedings of the 4th Spanish Conference on Information Retrieval*. CERI '16. Granada, Spain: Association for Computing Machinery, 2016. ISBN: 9781450341417 (cit. on p. 12).

[Muş+14] K. Muşlu, C. Bird, N. Nagappan, and J. Czerwonka. "Transition from Centralized to Decentralized Version Control Systems: A Case Study on Reasons, Barriers, and Outcomes". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 334–344. ISBN: 9781450327565 (cit. on p. 7).

[NC99] J. C. Nordbotten and M. E. Crosby. "The effect of graphic style on data model interpretation". In: *Information Systems Journal* 9.2 (1999), pp. 139–155 (cit. on p. 22).

[Num] NumPy. *NumPy*. URL: https://numpy.org/ (visited on 2023-08-13) (cit. on p. 35).

[OAL19] J. Ott, A. Atchison, and E. J. Linstead. "Exploring the applicability of low-shot learning in mining software repositories". In: *Journal of Big Data* 6.1 (2019), pp. 1–10 (cit. on p. 12).

[Pad12] B. Padmanabhan. "Unified Modeling Language (UML) Overview". In: *EECS810–Principles of Software Engineering* (2012) (cit. on p. 1).

[Pan] Pandas. *pandas*. URL: https://pandas.pydata.org/ (visited on 2023-08-13) (cit. on p. 35).

[Par] V. Paradigm. *What is Unified Modeling Language (UML)?* URL: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/ (visited on 2023-09-24) (cit. on p. 1).

[Pat03] M. Patrignani. "Visualization of Large Graphs". PhD thesis. Ingegneria Informatica, Univ. degli Studi di Roma, 2003 (cit. on p. 19).

[Per19] E. Persson. *Evaluating tools and techniques for web scraping*. 2019 (cit. on pp. 13, 14).

[Pet95] M. Petre. "Why looking isn't always seeing: readership skills and graphical programming". In: *Communications of the ACM* 38.6 (1995), pp. 33–44 (cit. on p. 18).

[Pro] G. Project. *GNU Wget*. URL: https://www.gnu.org/software/wget/ (visited on 2023-04-26) (cit. on p. 14).

[Pyt] Python. *What is Python? Executive Summary*. URL: https://www.python.org/doc/essays/blurb/ (visited on 2023-08-15) (cit. on p. 33).

[Ho-+14] T. Ho-Quang, M. R. Chaudron, I. Samúelsson, J. Hjaltason, B. Karasneh, and H. Osman. "Automatic Classification of UML Class Diagrams from Images". In: *2014 21st Asia-Pacific Software Engineering Conference*. Vol. 1. 2014, pp. 399–406 (cit. on p. 12).

[Ho-+17] T. Ho-Quang, R. Hebig, G. Robles, M. R. Chaudron, and M. A. Fernandez. "Practices and Perceptions of UML Use in Open Source Projects". In: *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. 2017, pp. 203–212 (cit. on pp. 11, 13).

[Qui03] P. T. Quinlan. "Visual feature integration theory: past, present, and future." In: *Psychological bulletin* 129.5 (2003), p. 643 (cit. on p. 18).

[Rea] G. V. Reasearch. *Version Control System Market Size, Share and Trends Analysis Report By Type, By Deployment, By Enterprise Size, By End Use, By Region, And Segment Forecasts, 2020 - 2026*. URL: https://www.grandviewresearch.com/industry-analysis/version-control-system-market (visited on 2022-11-25) (cit. on p. 7).

[RA16] A. Renkl and R. K. Atkinson. "Structuring the transition from example study to problem solving in cognitive skill acquisition: A cognitive load perspective". In: *Cognitive Load Theory*. Routledge, 2016, pp. 15–22 (cit. on p. 20).

[Req] Requests. *Requests: HTTP for Humans*. URL: https://requests.readthedocs.io/en/latest/ (visited on 2023-08-14) (cit. on p. 33).

[Resa] G. Research. *Google Colab - Frequently Asked Questions*. URL: https://research.google.com/colaboratory/faq.html (visited on 2023-08-13) (cit. on pp. 34, 35).

[Resb] G. Research. *Google Colaboratory*. URL: https://colab.google/ (visited on 2023-08-13) (cit. on p. 34).

[Rob+09] G. Robles, J. M. González-Barahona, D. Izquierdo-Cortazar, and I. Herraiz. "Tools for the study of the usual data sources found in libre software projects". In: *International Journal of Open Source Software and Processes (IJOSSP)* 1.1 (2009), pp. 24–45 (cit. on p. 26).

[Rob+17] G. Robles, T. Ho-Quang, R. Hebig, M. R. Chaudron, and M. A. Fernandez. "An Extensive Dataset of UML Models in GitHub". In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 2017, pp. 519–522 (cit. on pp. 11, 26).

[Sav+22] M. Savary-Leblanc, X. L. Pallec, P. Palanque, C. Martinie, A. Blouin, F. Jouault, M. Clavreul, and T. Raffaillac. "Mining Human Factors General Trends from +100k UML Class Diagrams". In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. MODELS '22. Montreal, Quebec, Canada: Association for Computing Machinery, 2022, pp. 913–922. ISBN: 9781450394673 (cit. on pp. 26, 27, 31).

[Sha48] C. E. Shannon. "A mathematical theory of communication". In: *The Bell System Technical Journal* 27.3 (1948), pp. 379–423 (cit. on p. 14).

[Sia04] K. Siau. "Informational and computational equivalence in comparing information modeling methods". In: *Journal of Database Management (JDM)* 15.1 (2004), pp. 73–86 (cit. on p. 20).

[SMP19] V. Singrodia, A. Mitra, and S. Paul. "A Review on Web Scrapping and its Applications". In: *2019 International Conference on Computer Communication and Informatics (ICCCI)*. 2019, pp. 1–6 (cit. on p. 13).

[Sir+15] D. S. Sirisuriya et al. "A comparative study on web scraping". In: (2015) (cit. on p. 13).

[Sli] Slintel. *Top 5 Version Control technologies in 2023*. URL: https://www.slintel.com/tech/version-control (visited on 2023-01-11) (cit. on p. 7).

[Soua] B. Soup. *Beautiful Soup Documentation*. URL: https://beautiful-soup-4.readthedocs.io/en/latest/ (visited on 2023-05-06) (cit. on p. 33).

[Soub] SourceForge. *SourceForge*. URL: https://sourceforge.net/ (visited on 2022-11-22) (cit. on p. 10).

[Sta73] H. Stachowiak. *Allgemeine modelltheorie*. Springer, 1973 (cit. on p. 5).

[Sta11] W. Stallings. *Network Security Essentials: Applications and Standards*. 4th. Prentice Hall, 2011. ISBN: 0136108059 (cit. on p. 9).

[Ste93] W. Steinmüller. "Informationstechnologie und Gesellschaft: Einführung in die angewandte Informatik". In: *(No Title)* (1993) (cit. on p. 5).

[Stö14] H. Störrle. "On the Impact of Layout Quality to Understanding UML Diagrams: Size Matters". In: *Model-Driven Engineering Languages and Systems*. Cham: Springer International Publishing, 2014, pp. 518–534 (cit. on pp. 23, 28–30, 62).

[Stö16] H. Störrle. "Diagram Size vs. Layout Flaws: Understanding Quality Factors of UML Diagrams". In: 2016-09, pp. 1–10 (cit. on pp. 2, 3, 23, 24, 29, 30, 43, 47, 95, 96).

[Ten] TensorFlow. *TensorFlow*. URL: https://www.tensorflow.org/ (visited on 2023-08-13) (cit. on p. 35).

[Tor] TortoiseSVN. *TortoiseSVN*. URL: https://tortoisesvn.net/ (visited on 2023-01-11) (cit. on p. 7).

[Tre82] A. Treisman. "Perceptual grouping and attention in visual search for features and for objects." In: *Journal of experimental psychology: human perception and performance* 8.2 (1982), p. 194 (cit. on p. 21).

[TG80] A. M. Treisman and G. Gelade. "A feature-integration theory of attention". In: *Cognitive psychology* 12.1 (1980), pp. 97–136 (cit. on p. 18).

[Tur+04] O. Turetken, D. Schuff, R. Sharda, and T. T. Ow. "Supporting systems analysis and design through fisheye views". In: *Communications of the ACM* 47.9 (2004), pp. 72–77 (cit. on p. 20).

[Win90] W. Winn. "Encoding and retrieval of information in maps and diagrams". In: *IEEE Transactions on Professional Communication* 33.3 (1990), pp. 103–107 (cit. on p. 17).

[Win93] W. Winn. "An account of how readers search for information in diagrams". In: *Contemporary Educational Psychology* 18.2 (1993), pp. 162–185 (cit. on pp. 18, 21, 23).

[ZND18] N. N. Zolkifli, A. Ngah, and A. Deraman. "Version Control System: A Review". In: *Procedia Computer Science* 135 (2018). The 3rd International Conference on Computer Science and Computational Intelligence (ICCSCI 2018) : Empowering Smart Technology in Digital Era for a Better Life, pp. 408–415. ISSN: 1877-0509 (cit. on p. 6).