An online corpus of UML Design Models : construction and empirical studies

Karasneh, B.H.A.

An Online Corpus of UML Design Models:
Construction and empirical studies

Bilal Karasneh

July 2016

- The author of this PhD thesis was partially financed by Erasmus Mundus program (JOSYLEEN)
- The author of this PhD thesis was employed at Leiden University

Typeset by LaTeX printed by Ipskamp Drukkers.
ISBN: 978-94-028-0242-9

An Online Corpus of UML Design Models:
Construction and empirical studies

Proefschrift

ter verkrijging van
de graad van Doctor aan de Universiteit Leiden,
op graag van Rector Magnificus Prof. Mr. C.J.J.M. Stolker
volgens besluit van het College voor Promoties
te verdedigen op donderdag 7 juli 2016
klokke 10:00 uur

door

Bilal Karasneh
geboren te Irbid, Jordan
in 1982

Promotiecommissie

Promotoren:

Prof. Dr. Joost N.Kok	Universiteit Leiden
Prof. Dr. Michel R. V. Chaudron	Chalmers Universitet and Göteborgs Universitet, Sweden

Commissieleden:

Prof. Dr. Ir. Thomas H. W. Bäck	Universiteit Leiden
Dr. Regina Hebig	Chalmers Universitet and Göteborgs Universitet, Sweden
Prof. Dr. Aske Plaat	Universiteit Leiden
Prof. Dr. Ir. Bedir Tekinerdogan	Wageningen University

To the spirit of my father Hikmat Karasneh,
and to my mother Maryam Obeidat

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, we discuss the context of the dissertation, the study motivation, and objectives. Also, the research approach and research methods are outlined.

Traditional engineering disciplines such as electrical and mechanical engineering are guided by physical laws. They constrain the engineering of solutions by enforcing regularity and limiting the complexity. Software engineering is not constrained by physical laws. Engineers often create complex software, which is difficult to understand, test and maintain. A large part of research in Software Engineering focuses on the software implementation because it is an inevitable part of software development. However, the implementation is a late activity of the software development process. Implementation is preceded by requirement and design. Software design is the backbone of the implementation. The quality of the design strongly influences the quality of the implementation. Solving problems during the design saves time, effort and cost of maintenance compared to solving the problems that grow out of design flaws only in the implementation. Software models are ways of expressing software designs. The Unified Modeling Language (UML) is used to develop and express software design. The benefit of designing software using a modeling language is that one discovers problems early. Also they provide a high-level overview of systems. Assessing the quality of software design models is an interesting research area. Design models should capture the requirements and also serve as a basis for the implementation. If a software model does not meet the requirements, it may lead to deliver a wrong product. The same applies when a software model

is difficult to understand, complex or difficult to implement, it is likely to lead to producing wrong software. In practice, the quality of software models is important for software architects, developers, testers and maintainers. Software developers, for example, need comprehensible models for producing implementation codes. The designs give developers a view of the software at a high abstraction, which highlights important classes and relationships between classes. Empirical studies are needed for defining, measuring, and testing of modeling quality. Empirical studies have become an important part of software engineering research and practice.

1.1 Problem Statement

Various views exist on what constitutes quality of software. One approach defines quality as the absence of defects. Conversely, one can see the amount and severity of defects as an indicator of the lack of quality in a software system. The assessing of severity of defects is currently mostly done manually. This is quite labor intensive, moreover it suffers from lack of agreement between different developers that perform the rating of defects. Because of the large effort involved in filling out defect reports, developers routinely fill out default values for the severity levels. We can summarize the problem as follows: software defects must be prioritized by assessing their severity. In this thesis, we explore how to automate severity assessment of software defects. Our interest in this thesis is in assessing the quality of software designs – esp. when represented using UML models. Ideally, we would study how UML models affect the quality of the final implementation in industrial projects. However, currently it is difficult to study UML models in an empirical manner because they are difficult to find and collect. Companies are extremely conservative in sharing their software designs because they see designs as part of their competitive advantage. However, we believe that a corpus of UML designs could be very beneficial to the SE research community. We foresee that a repository of examples of software design models can be useful to help to create better designs.

1.2 Objective of the Study

For the software defect severity, our aim is to research how this severity prediction can be achieved through reasoning about the requirements and the design of a system. For this purpose, we want to use ontologies to link reasoning about defects to knowledge about the requirements and design of the system. To enable more empirical studies about UML models, our aim is to build a repository that contains a huge number of UML models. We aim to collect these from the internet, literature and collaboration with companies and universities.

To guide our objectives, we formulate the following research questions:

- **RQ1**: How we can use knowledge about the requirements and design of a system to enable the automatic classification of defects into severity levels?
- **RQ2**: How we can establish a repository of UML models?
- **RQ3**: How can a repository of UML models be used for empirical studies?
- **RQ4**: Does the aid of a repository of software design models improve the quality of software designs created by novice designers?
- **RQ5**: What kind of design flaws can be detected in UML design models?

1.3 Research Methodology

Empirical research is a way of obtaining knowledge using direct and indirect observation or experience. Empirical studies attempt to compare theories with reality and improve the theories as a result. Empirical studies have become an important part of software engineering research and practice. 20 years ago, it was rare to see a conference or journal article about a software development tool or process that had empirical data to back up the claim. Nowadays, it is becoming more common that software engineering conferences and journals stimulate articles that describe a study or evaluation. Researchers in software engineering have been increasingly interested in empirical studies because it allows them to evaluate and improve techniques, methods and tools in developing software. Several methods are used to accomplish our research, including: case studies, experiments, and surveys. For the automated measures of defect severity, we use an industrial case studies that follow the same approach: data collection, data analysis and conversion, and data classification. More details about the methodology and the approach are in chapter 3. Collecting UML models from the internet and via collaboration with other universities is a type of field-study, and we use our collection. We ran experiments using the model repository to answer the research questions. We use a survey method for getting feedback about using our repository of models. Lastly, in our research journey we performed experiments for detecting design flows in UML class diagrams. We focus on design anti-patterns, where an anti-pattern is a literary form that describes a bad solution to recurring design problems that lead to negative effects on code quality.

1.4 Contributions

Our research aims to contribute to the area of software quality. More specifically, our contributions are in three areas: First, the area of automatic prediction of software defect severity. Secondly, the creation of UML Repository. Thirdly, the topic of quality

of UML-based software designs. The contributions in the first area of defect severity are:

1. A method that shows how knowledge about the software requirements and design can be captured and used to predict the severity levels of defects. Including knowledge of the requirements for establishing the severity of defects helps to reflect what is important according to the users of the system, not only according to its developers. This method leads to a better classification of severity of defects than is currently produced by professionals in industrial projects.

2. The use of ontologies and ontology reasoning (AI techniques) to capture and link knowledge about requirements and design. To this end, we propose a set of general rules for reasoning that is synthesized based on industrial projects.

3. The explicit representation of knowledge about classifying defects enhances the understandability of the knowledge that is often implicit and thereby enables the transfer and reuse of domain knowledge.

The contributions in the second area of quality of creation of UML repository are:

1. Creation of a repository with many UML software designs. This repository can be used as:

   (a) A source for experimental material for empirical studies of UML diagrams.

   (b) A basis for corpus studies related to UML modeling (e.g. benchmarking).

   (c) A source of examples of UML that can be used for learning UML by examples.

   (d) A start of an open community on UML modeling.

The contributions in the third area of quality of UML-based software designs are:

1. Provide empirical evidence about the benefits of creating and improving UML models with the aid of examples.

2. Provide empirical evidence of the difference between experts' and novices' assessment of the quality of UML models.

3. Provide a solution for detecting code smells and design anti-patterns in UML class diagrams. This contrasts with the current research which focusses on detecting anti-patterns in the source code implementation.

4. We found that the anti-patterns that occur in the design models, if not tended to, percolate to the source code where they decrease software quality.

1.5 Dissertation Outline

The outline of this work is as follows:

- **Chapter 2: Background.** Defines the foundation on which this dissertation is built and introduces the used terminology. This chapter discusses: (i) software defects and standards of software reports defects. (ii) Ontologies and ontology reasoning. (iii) Introduce UML as modeling language and elaboration of some UML diagram types that are commonly used in practice. (iv) Challenges in modeling using UML especially for novices, and challenges for applying empirical study on UML.

- **Chapter 3: A Method for Automated Prediction of Defect Severity Using Ontologies.** In this chapter we present our approach for creating and using ontologies for predicting the severity of software defects. In this study we use two case studies with 80 defects in total with different severity levels. We show how the data was collected from the company, how we analyze it and convert it to IEEE standard software anomalies report. We use five severity levels only. We show the results and a comparison between our results and using machine learning for predicting severity of the software defects. We also show the feedback from a software architect, software developer, software engineer and the project service coordinator, who emphasized that our approach yields very promising results.

- **Chapter 4: Establishing an Infrastructure for Empirical Research on UML Diagrams.** In this chapter we introduce: i) a crawler for collecting UML models from the internet, ii) a classifier for UML class diagram, iii) a tool (called Img2UML) that converts UML class diagram, sequence diagram and use case diagrams from image formats into XMI format. The Img2UML tool also extracts models information from images and stores this in a database. We show results of the tool, its limitation and some statistics related to the class diagrams collected from the internet.

- **Chapter 5: Models-db.com: An Online Repository UML Models.** In this chapter we present an online repository for UML models, especially with UML class diagrams. The repository contains images, XMI, and design metrics for a large number of class diagrams. The repository is searchable, and the user can search based on names of classes, attributes and operations. Furthermore, the repository allows users to share their models, to define experiments and to generate reports (including statistics and graphs). This repository will be useful for research as the first corpus of UML models. This repository will provide a good place for researchers and students to study and analyze UML models. This improves the possibilities for empirical studies in UML. Also, it supports the application of UML in education and industry.

- **Chapter 6: UML repository as benchmark for Quality Analysis.** In this chapter, we present a statistical analysis of models in the repository, and show some interesting patterns.
- **Chapter 7: Quality assessment of UML Class Diagrams.** We present the experiment we have conducted for comparing how experts and students assess different quality properties of class diagrams. Six quality attributes were addressed. The results reveal that the assessments of students and experts differ in all quality attributes, and that the experts are harsher in their evaluation.
- **Chapter 8: Using Examples for Teaching Software Design.** The goal of this research is to study the effects of using a collection of examples for creating a software design. We performed a controlled experiment for evaluating the use of a broad collection of examples for creating software designs by software engineering students. In this study, we focus on software designs as represented through UML class diagrams. The treatment is the use of the collection of examples. These examples are offered via our searchable repository.
- **Chapter 9: Conclusion and Future Work.** In this chapter we draw conclusions and discuss future work.

1.6 Publications

This is a chronological list of publications that were (co-)authored during this doctoral research:

1. M. Iliev, B. Karasneh, M. R. V. Chaudron, and E. Essenius, "Automated prediction of defect severity based on codifying design knowledge using ontologies," in *1st International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE 2012)*, pp. 7–11, June 2012 (Chapter 3)
2. B. Karasneh and M. R. V. Chaudron, "Extracting uml models from images," in *5th International Conference on Computer Science and Information Technology (CSIT2013)*, pp. 169–178, IEEE, 2013 (Chapter 4)
3. B. Karasneh and M. R. V. Chaudron, "Img2uml: A system for extracting uml models from images," in *39th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2013)*, pp. 134–137, IEEE, 2013 (Chapters 4 and 5)
4. B. Karasneh and M. R. V. Chaudron, "Online img2uml repository: An online repository for uml models.," in *Proceedings of the 3rd International Workshop on Experiences and Empirical Studies in Software Modeling (EESSMOD@MoDELS 2013)*, pp. 61–66, 2013 (Chapters 5, 6, 7 and 8)

5. T. Ho Quang, M. R. V. Chaudron, I. Samúelsson, J. Hjaltason, B. Karasneh, and H. Osman, "Automatic classification of uml class diagrams from images," in *Proceedings 21st Asia-Pacific Software Engineering Conference (APSEC 2014)*, 2014 (Chapters 4 and 5)

6. B. Karasneh, D. Stikkolorum, E. Larios, and M. R. V. Chaudron, "Quality assessment of uml class diagrams: A study comparing experts and students," in *MoDELS*, 2015 (Chapter 7)

7. D. Stikkolorum, T. Ho Ho Quang, B. Karasneh, and M. R. V. Chaudron, "Uncovering students' common difficulties and strategies during a class diagram design process: an online experiment," in *MoDELS*, 2015

8. B. Karasneh, R. Jolak, and M. R. V. Chaudron, "Using examples for teaching software design," in *Proceedings of the 22st Asia-Pacific Software Engineering Conference (APSEC2015)*, 2015 (Chapter 8)

9. B. Karasneh, M. R. V. Chaudron, F. Khomh, and Y.-G. Guéhéneuc, "Studying the relation between anti-patterns in models and in source code," in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)*, 2016 (Chapter 6)

Chapter 2

Background

In this chapter, we briefly introduce some software quality models. Then we introduce in short Ontologies development, languages, and reasoners. Later, we introduce UML as a software modeling language, and the UML diagram types that are commonly used. Finally, we explain the severity of software defect.

Software Engineering (SE) as defined in [10], is the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software. Nowadays many application domains demand reliable and sustainable software products. In order to be able to discuss and measure software quality, several quality models have been proposed. We will discuss these in this chapter. Also, we will discuss the notion of software defect and defect severity. Next, we will introduce the concept of the UML notation which is used for representing software designs. We conclude this chapter with discussing the main concept of ontologies. These will form the basis for representing knowledge about software systems and their application domains.

2.1 Quality Models

Software quality models are a set of software quality characteristics (also called quality attributes and quality properties) and their associations. These characteristics are quantifiable so that it provides the basis for assessing the quality of software systems. The effort for assuring that software is going to have certain quality attributes called

Software Quality Assurance (SQA). SQA defines a set of activities and procedures to control a product during its development, which at the end possess the expected quality attributes.

2.1.1 Software Quality Models

There are several software quality models presented to assess object-oriented quality attributes. The assessments are quantitatively (e.g. using metrics), or qualitatively through informal assessments, such as peer review. We show some renowned quality models:

- ***ISO/IEC 25010*** [11]: this standard is the replacement of the standard ISO/IEC 9126 [12]. Compatibility was added as a main characteristic, and security moved from a sub-characteristic to the main characteristic of its set of sub-characteristics. Some other sub-characteristics were added in this revision (confidentiality, integrity, nonrepudiation, accountability and authenticity, functional completeness, capacity, user error protection, accessibility, availability, modularity and reusability), while compliance was removed. Figure 2.1 illustrate the ISO 25010 quality model.

- ***McCall's Model*** [13]: This is the first quality model introduced in 1977. The authors differentiate between two quality attributes known as quality factors. The second level of quality attributes known as quality criteria that can be measured.

- ***Boehm Models*** [14]: The Boehm tries to overcome the problems of McCall's model, and it addresses the shortcomings of evaluating the quality of software. It added some more characteristics to McCall's model, emphasizing maintainability and hardware performance. It presents a hierarchical structure for high-level, intermediate level and primitive characteristics.

- ***QMOOD Model*** [15]: Bansiya and Davis introduced a hierarchical quality model for object-oriented systems based on Dromey's Model (MOOD) [16]. The model defines evaluation functions for such quality attributes as reusability, flexibility and understandability, based on eleven object-oriented design metrics. However, it does not formally define metrics.

- ***PQMOD Model*** [17]: This quality model is composed of a set of rules for the evaluation of quality taking in account design patterns.

- ***Lange and Chaudron*** [18]: To the best of our knowledge, this is the only specific model for quality of UML models. An overview of their framework is in Figure 2.2. This quality model is different from other models in that it considers UML models as an intermediate product of software development that derives it quality from the degree by it supports other software engineering activities.

Figure 2.1: *Framework of ISO 25010 quality models*

2.1.2 Measuring Software Quality

The measurement lies at the heart of many systems in our lives. Economic measurements determine price and pay increases. Medical system measurements enable doctors to diagnose specific illnesses. Measurements in atmospheric systems are the basis for weather prediction. Therefore, measurement helps us to understand our world, interact with our surroundings, and improve our lives. Fenton [19] shows that in software engineering, measurement is important for three activities: First, the measurement can help us to *understand* what is happening during development and maintenance. We assess the current situation, establishing baselines that help us to set goals for future behavior. Second, the measurement allows us to *control* what is happening in our projects. Using our baselines, goals, and understanding of relationships, we predict what is likely to happen and make changes to processes and products that help us to meet our goals. Third, measurement encourages us to *improve* our processes and products. For instance, we may increase the number or type of

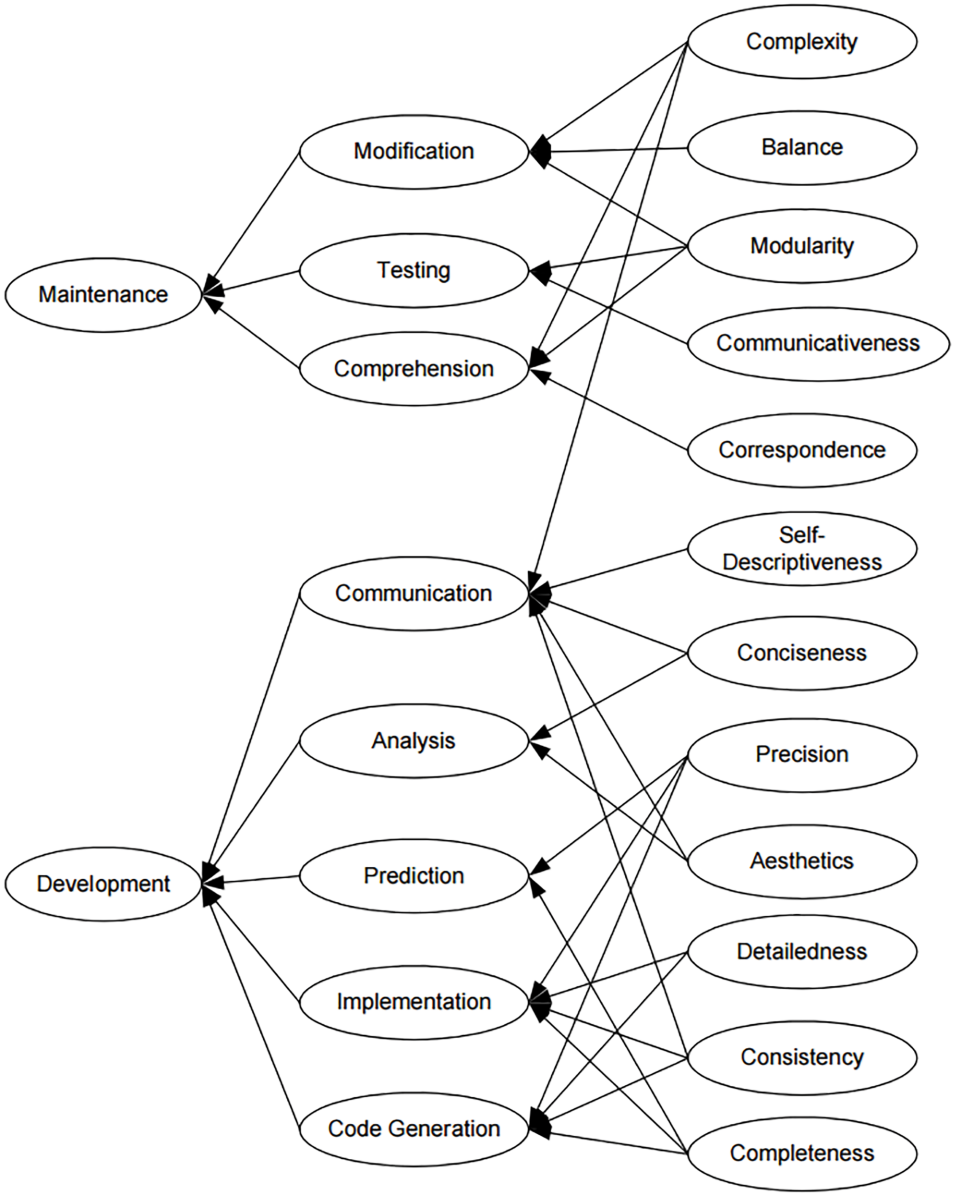

Figure 2.2: *Framework of (Lange and Chaudron) for quality of UML models*

design reviews we do, based on measures of specification quality and predictions of likely design quality. Measuring an entity is measuring the attributes of that entity. Understanding the attributes of an entity helps to understand the entity better. Design

measurement is the application to measure design artifacts. It aims at understanding, predicting, controlling or improving the quality attributes of the software product. For measuring software design, the practices have been revolving around the use of metrics [20]. Design metrics are used to assess the quality, size and complexity of software. They look at the quality of the software design at a particular point in the development cycle. Design metrics tend to be more locally focused and more specific, thereby allowing them to be used effectively to examine directly and improve the quality of the product's components. The most famous design metrics originate from the work of Chikdamber and Kemerer [20]. They developed six object-oriented design metrics that are still widely used in various design measurement nowadays. Many works in software quality prediction aim to predict the probability of software model to contain faults [21][22][23]. Most of these work validate metrics proposed in [20] for their effects or relation to quality aspects of software such as module fault-proneness [24].

2.2 Severity of Software Defect

According to the IEEE Standard Classification, for Software Anomalies [25], the cause of a software problem is called a software defect. We show the common vocabulary for terms useful in this context:

- ***Defects:***
  - A fault if it is encountered during software execution (thus causing a failure) [25].
  - Not a fault if it is detected by inspection or static analysis and removed prior to executing the software [25].
- ***Fault:*** an incorrect step, process, or data definition in a computer program [10].
- ***Failure:*** represents the inability of a system or component to perform its required functions within specified performance requirements [10].
- ***Error:*** A human action that produces an incorrect results [25].

The dictionary in [10], relates all these terms to one another by distinguishing between:

- a human action (a mistake),
- It is manifestation (a hardware or software fault),
- The result of the fault (a failure),
- The amount by which the result is incorrect (the error).

Hence, a software defect is the reason for producing an incorrect or unexpected result in a computer program or system, or it causes it to behave in unintended ways. Therefore, to deploy a high-quality software product, it needs to be tested first. Defects found in the testing phase need to be solved within a specific time constraint – before the deployment date. Software teams need to decide on the order in which to fix these defects. The assignment of severity levels to defects is specific for every software system or company and is done manually, usually by test analysis according to their expertise. However, it is often the case that a defect is assigned the default severity level, which typically is medium. A user might not agree with the assignment of the default severities level and might want to fix some defects sooner than others. In the next chapter, we explain how we use ontologies to automate assigning severity levels to software defects.

2.3 Ontologies in SE

The most common definition of ontologies says that an ontology is an explicit specification of a conceptualization [26]. In other words, ontologies are explicit formal specifications of the terms in the domain and the relations among them [26]. According to a more elaborate version of the definition, an ontology defines a common vocabulary for researchers who need to share the information in a domain. It includes machine-interpretable definitions of basic concepts in the domain and relations among them [27]. Moreover, ontologies formalize knowledge, represented in a language that supports reasoning [28]. Developing an ontology is similar to defining a set of data and their structure to be used by other programs. For instance, problem-solving methods, domain-independent applications, and software agents use ontologies and knowledge bases built from ontologies as data [27]. We summarize some reasons for developing ontologies:

- To share a common understanding of the structure of information among people or software agents. to enable reuse of domain knowledge.
- To make domain assumptions explicit.
- To separate domain knowledge from the operational knowledge.
- To analyze domain knowledge.

2.3.1 Ontology Editors

Developing an ontology requires a specialized environment for editing that makes it easier to build and maintain them. Such environments are called ontology editors. Currently, there are many ontology editors, each having its strengths and weaknesses.

According to the World Wide Web Consortium (W3C)[1] , examples of ontology editors are Protégé[2], SWOOP[3], OntoStudio[4], NeOn Toolkit[5], Knoodl[6]. In addition to an editor, a reasoner is useful to enable automated reasoning about the ontology.

2.3.2 Web Ontology Language

Web Ontology Language (OWL)[7] is the most recent development in standard ontology languages. OWL is a W3C Recommendation for representing ontologies, and it is the language with the strongest impact on the Semantic Web [29]. OWL is intended to provide a language that can be used to describe classes (concepts) and the relations between them that are inherent in Web documents and applications. The logical model is the base of OWL, which makes it possible for concepts to be defined and described. Complex concepts can be built up out of simpler concepts. Moreover, the logical model allows the use of a reasoner, which can help to maintain the hierarchy of the concepts correctly [30]. As explained in the OWL Guide[8] and at [30], OWL provides three sublanguages: OWL-Lite, OWL-DL, and OWL-Full. All of these are designed for use by specific communities of implementers and users. The defining feature of each sublanguage is its expressiveness. OWL-Lite is the least expressive while OWL-Full is the most expressive. OWL-DL's expressiveness falls in between. Each sublanguage is an extension of its simpler predecessor, both in what they can legally express and in what can validly conclude. OWL-Lite is the sublanguage with the simplest syntax. Its intended use is in situations where only a simple class hierarchy and simple constraints are required [30]. Because of the simple class hierarchy and constraints, automated reasoning is not used in OWL-Lite ontologies. OWL-DL is more expressive than OWL-Lite. OWL-DL is intended to be used when users want the maximum expressiveness without losing computational completeness[9], and decidability[10] of reasoning systems. OWL-DL is so named because it is based on Description Logics (DL). According to [30], Description Logics represent a decidable fragment of First Order Logic and are amenable to automated reasoning. Therefore, it is possible to compute the classification hierarchy automatically and check for inconsistencies in an ontology that conforms to OWL-DL [30]. OWL-Full is the most expressive sublanguage. It is meant for users who want maximum expressiveness with no guarantees for decidability or computational completeness. Hence, it is not possible to perform automated reasoning on OWL-Full

[1] http://www.w3.org/
[2] http://protege.stanford.edu/
[3] http://www.mindswap.org/2004/SWOOP/
[4] http://www.ontoprise.de/en/products/ontostudio/
[5] http://neon-toolkit.org/
[6] http://www.knoodl.com/
[7] http://www.w3.org/2004/OWL/
[8] http://www.w3.org/TR/owl-guide/
[9] All entailments are guaranteed to be computed
[10] All computations/algorithms will finish in finite time

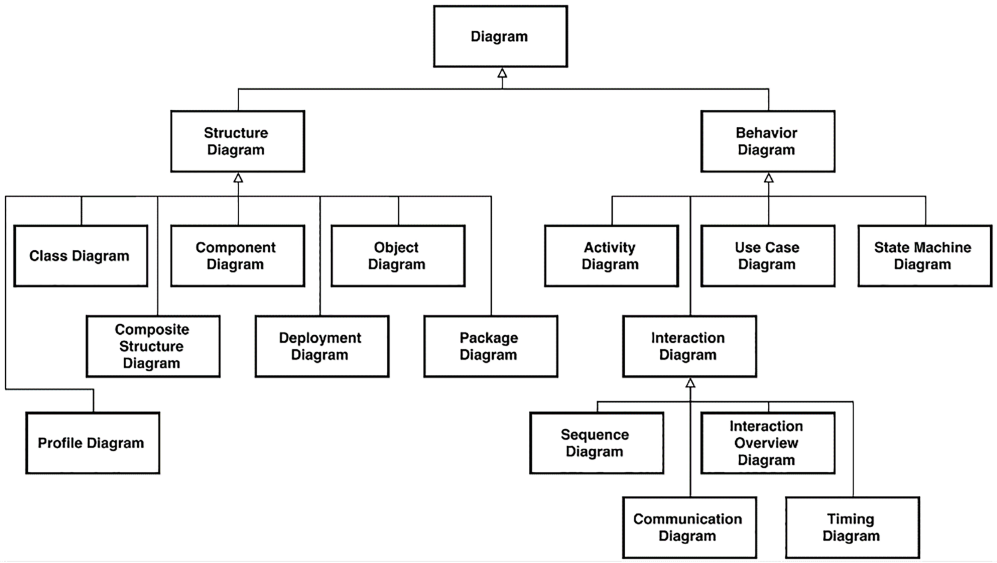

Figure 2.3: *The Taxonomy of UML Diagram Types*

ontologies, as stated in [30].A reasoner (also called inference engine) is a software application that derives new facts or associations from existing information [**?**]. It is a key component for working with ontologies. The survey results in [**?**] indicate that the most popular reasoners are Jena[11], RacerPro[12], Pellet[13] and FaCT++[14]. We have chosen the Pellet reasoner. It supports the full expressivity of OWL-DL and satisfies our reasoning needs.

2.4 Unified Modeling Language

Unified Modeling Language (UML) is a standard modeling language that created in 1997 by the Object Management Group[15]. Since then, it has been the industry standard for modeling software-intensive systems. Figure 2.3 shows the taxonomy of the UML diagrams.

UML was born out of the object modeling technique (OMT) [31], Booch[32], and Object-Oriented Software Engineering (OOSE) [33]. UML nowadays the de facto standard for software industry [34]. The current standard of UML, i.e., when we write this thesis, is version 2.4, which was released in August 2011. A beta version 2.5

[11] http://jena.apache.org/about_jena/about.html/
[12] http://www.racer-systems.com/products/racerpro/
[13] http://pellet.owldl.com/
[14] http://owl.man.ac.uk/factplusplus/
[15] http://www.omg.org/

released in September 2013. In UML 2.4, there are 14 types of diagrams divided into three categories, structure diagrams, behavior diagrams and interaction diagrams as shown in Figure 2.3. From the 14 types of diagrams, three diagrams are the most used in practice, namely class diagram, sequence diagram, and use case diagram [35]. We briefly introduce these diagrams next.

2.4.1 Class Diagrams

The class diagram is the most common structural model of the UML. Class model represents the static structure of the system in terms of classes, relationships between these classes and constraints in the relationships. The class diagram also constrains the way classes may interact with each other.

2.4.2 Sequence Diagrams

UML sequence diagrams are used to model the interaction behavior of systems. The sequence diagram shows the interactive behavior of collaborations of interaction participants working together by depicting the sequence in which messages exchange.

2.4.3 Use Case Diagrams

Use case diagrams capture the functionality of software system by describing which interactions should be supported between users and the system. It contains use cases, actors, and their relationships.

2.4.4 Challenges of modeling and studying using UML

UML offers flexibility and freedom in modeling. There is no one correct design for a given problem, and different correct scenarios can be proposed. There is a need to assess the quality of UML models to differentiate between solutions. For this, we need to study UML models deeply. One of the main problems of studying UML models is the lack of sharable software development software [36]. The collection of models from commercial software development is difficult because for different reasons companies like to keep their system design confidential. In open source software, development use of UML is not as common as the (inevitable) use of source code. Therefore, in Software Engineering there is a need to share modeling artifacts [37]. Therefore, collecting UML models is more difficult, and this difficulty makes empirical research of UML challenging. Moreover, there is no open technology for creating model repositories as there exist for source code. Many free code repositories are available, which improves the ability to develop code metrics, and facilitates empirical research for source code domain in general.

One problem that makes collecting of UML models challenging is the large variety of representations by different Computer-Aided Software Engineering (CASE) tools. These differ in both graphical representation and/or in terms of XML Metadata Interchange (XMI).

We found that UML models are available in abundance on the Internet, but rather than in CASE-tool format, they are stored in image formats. The problem with image formats is that the model content (e.g. class names) cannot be extracted out of them. Although many CASE tools support features like creating, modifying and exporting UML models into different formats, current CASE tools cannot recognize UML in images. This inability of CASE tools limits the usability of the UML models that are available as images.

2.5 Repositories in SE

Creating repositories is common in different domains, and it is important to preserve the history and the evolution of the collected data for future use, especially in research. In addition, repository-managers manage the accessibility of the available data. Repositories can be classified in many different ways including but not limited to:

- Types of data, such as PDF, images and videos.
- Contents of data, such as newspapers, sports and medicine.
- Technology used, such as rational database and file system.
- Users, such as students, researchers and fans.

We can classify repositories based on data available into two general categories: Disciplinary repositories and Multidisciplinary repositories.

Disciplinary repositories are repositories that archive works and data associated with these works in a particular subject area. For example, in biology, bio-repositories are important because they maintain biological samples, and preserve samples and assure the quality of these samples. Specimen Central[16] is the world's open biospecimen research database. Another example is in the area of linguistics, SLDR[17] is a speech and language data repository that gathering and sharing language data.

Multidisciplinary repositories are repositories that archive works related to different subjects. ELSEVIER[18] and nature.com[19] are examples of these repositories, where they have many articles related to different research areas.

[16] http://specimencentral.com/
[17] http://sldr.org/
[18] https://www.elsevier.com/
[19] https://www.nature.com/

In software engineering, software systems become more complex, and produce a large number of artifacts from documentation and different kind of models to the source code. It is difficult task to organize and share these artifacts because it contains different material types. Therefore, many repositories have been created for a different purpose.

In addition, many conferences are held to propose new repositories, challenging, data showcase and experiments on the available dataset. For example Mining Software Repositories (MSR) and PRedictOr Models In Software Engineering (PROMISE) conferences. MSR is an international conference and is co-located with International Conference on Software Engineering (ICSE) since 2004. MSR field analyzes the rich data in software repositories to discover interesting information about software systems.

Rodriguez et al. [38] classify repositories in software engineering as well as discussing their open problems. They classify software engineering repositories into:

1. Source code, can be used to study software properties, such as size and complexity.
2. Source Code Management Systems, it stores all the changes that the different source code undertake during the project.
3. Issue tracking system. Bugs, defects and user requests are archived in issue tracking systems, where users and developers can meet and discuss about defects found, or new functionality.
4. Messages between developers and users. The messages between users and developers are archived in the form of mailing lists, which can also be mined for research purposes.
5. Meta-data about the projects. This meta-data may include intended-audience, programming language, domain of application or license.
6. Usage data. For example, statistics about software downloads.

We show that Rodriguez et al. [38] are missing models repository, which are repositories contains software design models.

Some CASE Tools have model repositories that enable collaborative modeling. This collaborative lets members commit and update models. This is the same as in the source code developments by version control systems. For example, collaborative modeling as in VisualParadigm[20], where it lets modelers work on the same project concurrently without overwriting each other's works. Actually, the use of standard version control systems such as Subversions (SVN) is not sufficient, because we need more such as searching models contents and collaboration.

We show some examples of software repositories that are proposed for different purposes:

[20] http://www.visual-paradigm.com/

- Repositories of source code:
  - CodeProject[21].
- Repositories of source code and support versioning:
  - GitHub[22].
  - Bitbucket[23].
  - Google code[24].
- Repositories of code metrics and defects:
  - PROMISE Repository[25].
- Repositories for design models:
  - ReMoDD repository[26].
  - VPository[27].

[21] http://www.codeproject.com/
[22] https://github.com/
[23] https://bitbucket.org/
[24] https://code.google.com/
[25] http://openscience.us/repo/index.html
[26] http://www.remodd.org/
[27] http://www.vpository.com/

Chapter 3

A Method for Automated Prediction of Defect Severity Using Ontologies

*In this chapter, we present **MAPDESO** – a Method for Automated Prediction of DEfect Severity using Ontologies. This method was developed based on industrial case studies. The method is based on classification rules that consider the software quality properties affected by a defect, together with the defect's type, insertion activity, and detection activity.*

This chapter is based on the following publication:

- Martin Iliev, Bilal Karasneh, Michel R.V. Chaudron, Edwin Essenius. **Automated prediction of defect severity based on codifying design knowledge using ontologies.** *In Proceedings of the First International Workshop on Realizing AI Synergies in Software Engineering (RAISE '12), pages 7-11, Zurich, Switzerland. 2012.*

As part of quality assurance in software development it is common to assign severity levels to defects. Whether a defect is of high of low severity is specific for every software system or company. The assignment of severities is mostly done manually, usually by test analysts who base this on their expertise. Different projects use different scales of severity levels. Common scales for defect severity contains three, four or five severity levels (sometimes even more).

A common, yet poor practice is that engineers do not take the assignment of severity level seriously. Very often, engineers use the default severity level, which typically is 'medium'. Moreover, engineers sometimes make mistakes when assigning severities. Overall, these factors lead to the assignment of wrong severity levels to defects. To address this problem, we have researched how to automatically predict the severity of defects. This prediction uses knowledge of the software development process while decreasing the workload of the software architects and the test analysts. We use this knowledge to assign severities that reflect what is important not only for the developers but also for the users. The aim is to devise a method for automatically predicting the severity levels of defects found during testing at the system level and also during coding and maintenance. We name our method MAPDESO – a Method for Automated Prediction of DEfect Severity using Ontologies. Such a method would be especially useful for medium-to-large software systems, where the probability of defects occurrences is more. Hence, there is fair to a large amount of effort involved in assigning defect severity levels and moreover, the problems such as poor prioritization of defects may be more severe in larger projects.

We compare the performance of MAPDESO with machine learning algorithms. We use Weka data mining software for using ten machine learning algorithms. We compare the result of MAPDESO and machine learning algorithms with the original (manual) classification from the defects report. The result of the comparison shows that MAPDESO performs better on the classification of severity levels.

3.1 Approach

This section contains the description of MAPDESO. MAPDESO has culminated in the development of an ontology for automated prediction of defect severity (automatic classification of defects into the severity levels from the IEEE standard in [25]). The process of developing the ontology is an essential part of MAPDESO. However, once the ontology is developed, this process does not need to be repeated when using the method. In other words, developing the ontology is done only once, while it can be used many times. In the beginning, we will explain how the ontology and the classification work by describing the process of developing the ontology. After that, we will explain the method flow. We will refer to using the ontology as a black box process – only the input, and the output will be mentioned.

3.1.1 Developing the Ontology

We selected the Protégé platform for ontology development because of its functionality and popularity. Protégé has proven to be the most popular and user-friendly – with a market share of 68.2%. Also, it has many available plug-ins [29][39]. We have chosen the Web Ontology Language (OWL) as the development ontology language.

In Chapter 2, we referred to an approach for ontology development. We will follow that approach when explaining how the ontology was developed.

3.1.1.1 Meta-meta Level

This is the phase for defining the foundation of the ontology. For our purpose, we can use the existing meta-meta model that comes with existing ontology-tools. We use Protégé-OWL as ontology editor, OWL and OWL-DL as ontology languages, and Pellet as a reasoner. Hence, the ontology development approach has a predefined meta-meta level.

3.1.1.2 Meta Level

This is the phase in which the key concepts in the ontology and their relations are defined. For our ontology, in this phase, we have defined and created the base classes and the properties. Classes are the focus of most ontologies, and they represent concepts in a domain of discourse [27]. They are described using formal (mathematical) descriptions that state precisely the requirements for membership of the class. They may be organized into a superclass-subclass hierarchy, also known as a taxonomy [30]. At this level of ontology development, we created the following classes:

- *Defect* – this class represents all defects.
- *Effect* – this class represents attribute Effect from the IEEE standard [25]. Its values are quality properties and classes of requirements that are impacted by a failure caused by a defect.
- *Type* – this class represents Attribute Type from the IEEE standard [25]. The type of a defect represents the nature of that defect. The attribute's values are categorizations based on the class of code or the work product within which a defect is found.
- *InsertionActivity* – this class represents attribute Insertion activity from the IEEE standard [25]. Its values are the activities during which a defect is inserted.
- *DetectionActivity* – this class represents attribute Detection activity from the IEEE standard [25]. Its values are the activities during which a defect is detected.

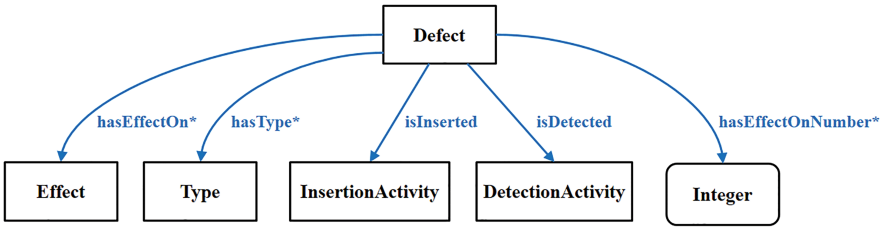

Figure 3.1: *The created classes and properties for the ontology*

We created the properties describing the relations between the defects and the attributes from the standard. These properties describe the relations between class Defect and classes *Effect*, *Type*, *InsertionActivity* and *DetectionActivity*.

The created classes and properties for the ontology are shown in Figure 3.1. There are five properties depicted in Figure 3.1. The property *hasEffectOn** is an object property linking an individual class to another individual class [30]. This property relates class *Defect* (domain of the property) to class *Effect* (its range). The *hasEffectOn** property relates a defect to one or more of its affected quality properties (e.g., performance, functionality). The asterisk at the end of the property means that its range accepts one or more values. Properties *hasType**, *isInserted* and *isDetected* can be explained in a similar way as the property *hasEffectOn** (these four properties are in blue). The last property shown in figure 3.1 is *hasEffectOnNumber* (depicted in black). This is a datatype property (linking an individual to a specific datatype [30], for example, integers) that relates class Defect and its subclasses (domain) to datatype Integer (range). This property represents the number of values (an integer) of attribute Effect that are affected by a defect. The asterisk means that its range accepts one or more values. Moreover, in Figure 3.1, the datatype Integer is given in a rounded rectangle to point out that it is not a class (depicted with rectangles) but a datatype.

3.1.1.3 Class Level

In this phase, we define more detailed items of information needed for our classification as well as their relation to the top-level concepts. For class Defect, we define the following sub-classes:

- *DefectWithBlockingSL* – this class represents all defects assigned blocking severity level.
- *DefectWithCriticalSL* – this class represents all defects assigned critical severity level.
- *DefectWithMajorSL* – this class represents all defects assigned major severity level.

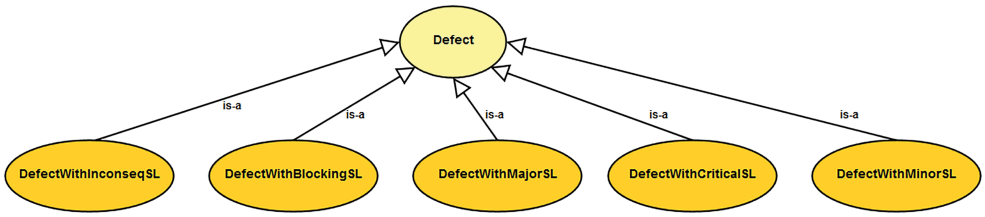

Figure 3.2: *Class Defect, its subclasses*

- *DefectWithMinorSL* – this class represents all defects assigned minor severity level.
- *DefectWithInconsequentSL* – this class represents all defects assigned inconsequential severity level.

These five classes that are related to the five severity levels from the IEEE standard [25]. These classes are defined as disjoint from each other because every defect is assigned one and only one severity level. The class hierarchy for class Defect and its subclasses is presented in Figure 3.2.

Next, we created the subclasses for the other four classes. Because the classes are the attributes from the IEEE standard [10], their subclasses are the values of the respective attributes. As both the attributes and their values are clearly listed, and 13 are defined in the IEEE standard [10], we are not going to repeat this information here. It should be noted though that the four classes we are referring to are *Effect*, *Type*, *InsertionActivity* and *DetectionActivity*, as given in Figure 3.1.

3.1.1.4 Instance Level

Instances represent knowledge/facts that is specific to real projects or systems. Hence, specific defects in the ontology can be regarded as instances.

3.1.1.5 Classification Rules

For our ontology, we have developed classification rules that handle the classification of the defect instances into one of the five severity levels from the standard. These classification rules are implemented in Class Description. In OWL, there are three main categories of restrictions: Quantifier Restrictions, Cardinality Restrictions and hasValue Restrictions [30]. Using these restrictions, we have developed five sets of rules – one set of rules for each of the five classes *DefectWithBlockingSL*, *DefectWithCriticalSL*, *DefectWithMajorSL*, *DefectWithMinorSL* and *DefectWithInconseqSL*. The classification rules represent necessary and sufficient conditions for a defect to belong to one and only one of the above five classes. In other words, if a defect satisfies the set of rules corresponding to one of the five classes, then this defect belongs to that class. Then it

Table 3.1: *Example of defects report of the project CS1 converted to IEEE standard [25]*

Defect ID	Effect	Type	Insertion Activity	Detection Activity	S. L. from the project	S. L. Converted to IEEE
101	Functionality; Security; Performance; Serviceability	Data; Interface	Design	Supplier testing	Show-stopper	Blocking
...	...	...	...	...	...	...
110	Functionality; Performance	Data; Logic	Configuration	Coding	Severe	Critical
...	...	...	...	...	...	...

is assigned the severity level corresponding to the class (i.e., blocking, critical, major, minor or inconsequential severity level).

The classification rules complement the developed ontology. Hence, it is important to point out that these rules were developed manually based on the pattern of the empirical data (from two case studies - see Section 3.2) and on heuristic strategies, such as intuitive judgment. The rules were later improved to be as general as possible to apply the method to various software projects (see the validation in Section 3.3). We give more weight to defects inserted during the requirements and design phases than during the coding and configuration phases - this way, the defects inserted earlier in the software cycle will be given higher severity levels and hence, fixed sooner than other defects. We consider the quality properties affected by a defect as a key component (but are not restricted only to that) for classifying the defect into one of the five severity levels. Thus, the greater the extent to which a defect affects the quality of the software, the higher the severity level that will be assigned to the defect. Table 3.1 shows examples of defect attributes and their values that are converted to the IEEE Standard in [25]. We list the rules in Table 3.2. We explain the meaning of the rules R1 and R2.

The sub-rules of R1 mean the following: an entity is assigned critical severity level if and only if it is:

(R1.1) a defect. (R1.2) affects exactly two or three of the values of attribute Effect. (R1.3) is inserted during the design phase or the requirements phase, or inserted during the coding phase or the configuration phase and affects exactly three values of attribute Effect or at least two values of attribute Type. (R1.4) affects one or more of the values Data, Interface or Logic of attribute Type. (R1.5) is detected during the coding phase,

Table 3.2: *Classification rules of detecting severity of defects*

• **Rule 1** (R1): defines the necessary and sufficient conditions for a defect with blocking severity level (class *DefectWithBlockingSL*). It consists of two sub-rules and they are the following: – (R1.1) *Defect* – (R1.2) *hasEffectOnNumber* min 4
• **Rule 2** (R2): defines the necessary and sufficient conditions for a defect with critical severity level (class *DefectWithCriticalSL*). It consists of five sub-rules and they are the following: – (R2.1) *Defect* – (R2.2) (*hasEffectOnNumber* exactly 2) or (*hasEffectOnNumber* exactly 3) – (R2.3) (*isInserted* only (*InDesign* or *InRequirements*)) or ((*isInserted* only (*InCoding* or *InConfiguration*)) and ((*hasEffectOnNumber* exactly 3) or (*hasType* min 2))) – (R2.4) *hasType* only (*Data* or *Interface* or *Logic*) – (R2.5) *isDetected* only (*FromCoding* or *FromSupplierTesting* or *FromCustomerTesting* or *FromProduction*)
• **Rule 3** (R3): defines the necessary and sufficient conditions for a defect with major severity level (class *DefectWithMajorSL*). It consists of two sub-rules and they are the following: – (R3.1) *Defect* – (R3.2) not *DefectWithBlockingSL* and (not *DefectWithCriticalSL* or ((*isInserted* only (*InCoding* or *InConfiguration*)) and (*hasEffectOnNumber* exactly 2) and ((*hasType* only *Data*) or (*hasType* only *Interface*) or (*hasType* only *Logic*)))) and not *DefectWithMinorSL* and not *DefectWithInconseqSL*
• **Rule 4** (R4): defines the necessary and sufficient conditions for a defect with minor severity level (class *DefectWithMinorSL*). It consists of four sub-rules and they are the following: – (R4.1) *Defect* – (R3.2) *hasEffectOn* some (not *Usability* and not *Security*) – (R4.3) *hasEffectOn* only (not *Usability* and not *Security*) – (R4.4) *hasEffectOnNumber* max 1
• **Rule 5** (R5): defines the necessary and sufficient conditions for a defect with inconsequential severity level (class *DefectWithInconseqSL*). It consists of three sub-rules and they are: – (R5.1) *Defect* – (R5.2) *hasEffectOn* some *Usability* – (R5.3) *hasEffectOn* only *Usability*

or the supplier-testing phase, or the customer testing phase, or during production use.

The sub-rules of R4 mean the following: an entity is assigned minor severity level if and only if it is: (R4.1) a defect. (R4.2) affects some property values of the Effect attribute except the Usability and Security. (R4.3) affects only property values of attribute Effect that are not Usability and Security. This sub-rule (R4.3) is needed to make sure that the defect can only have the specified values. Such a sub-rule is known as a closure axiom [30]. (R4.4) affecting exactly one property value of attribute Effect. The classification rules complement the developed ontology.

3.1.2 The Method Flow

In this subsection, we will focus on how to use the ontology to automatically classify the severity levels of defects from different projects. The method consists of the following steps:

1. detecting defects;
2. analyzing and converting information about the defects into the IEEE Standard;
3. input the converted information into the ontology;
4. predicting the severity levels of the defects.

These steps are illustrated in the activity diagram in Figure 3.3. The diagram represents a reference for the description of the method flow. As illustrated in Figure 3.3 MAPDESO can be used in two different settings. The first option is to apply the method to a project that does not use the IEEE standard [25] for describing its defects. The second option is to apply the method to a project that has adopted the IEEE standard [25] for describing its defects. The difference is the omission of one-step from the method as given in the figure. The reason stems from the fact that once a project is using the IEEE standard [25] for describing its defects, then the defects and their information can be directly input in the ontology. There is no need to convert the defects' information because it is already in the form needed to enter the defects into the ontology.

3.2 Case Studies

We conducted two case studies in an industrial environment: the Technical Software Engineering Practice of Logica[1]. Both cases studies follow the same approach, as depicted in Figure 3.3. The approach consists of three parts: data collection, data analysis and conversion, and data classification. As mentioned in Section 3.1.1.5, the classification rules were developed using the data from these two case studies. Thus, this data can be regarded as the training data for the developed ontology and rules.

[1]Logica is a Software Development Company and has been acquired by CGI

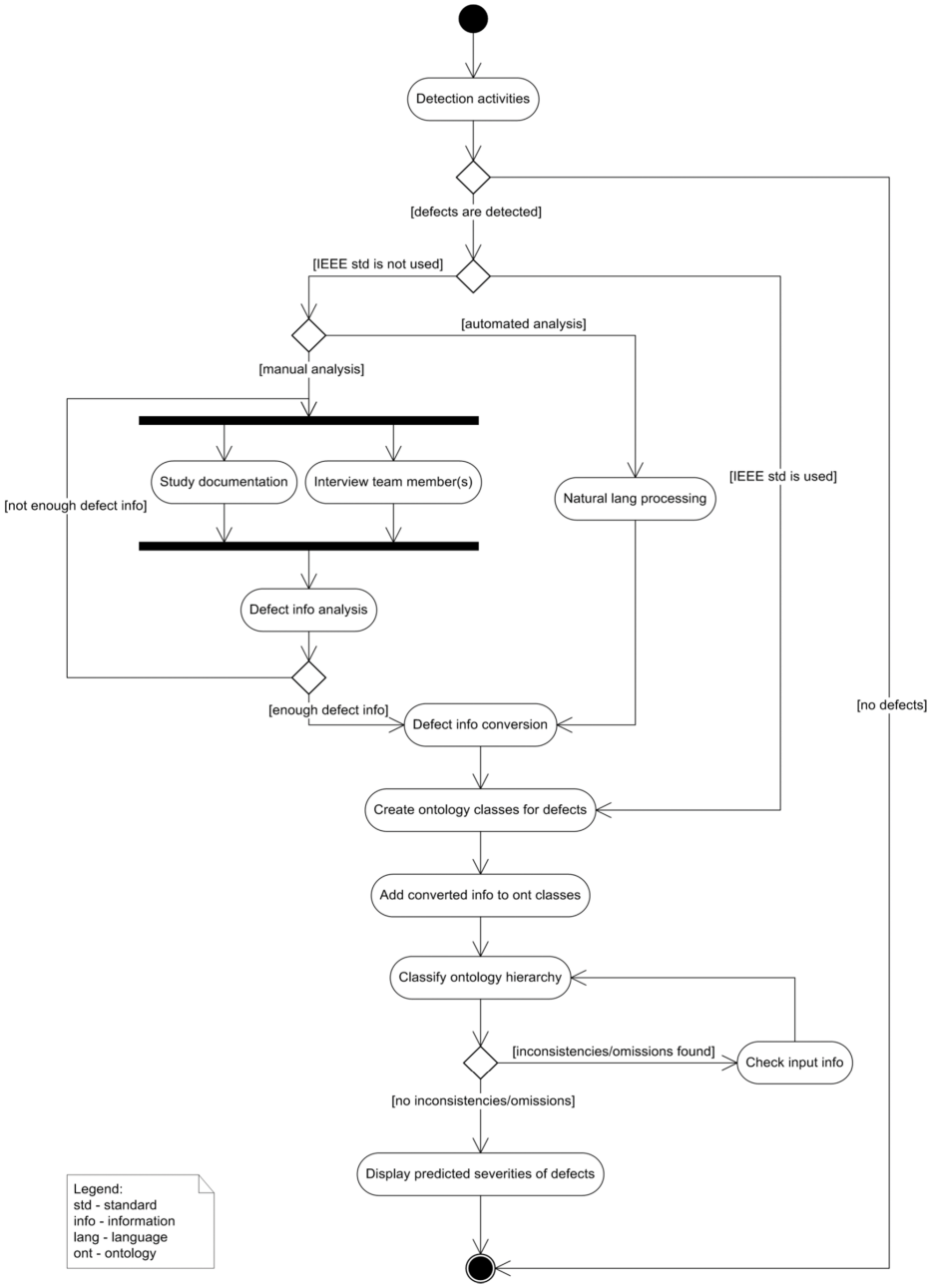

Figure 3.3: *Activity diagram for the prediction of defects' severity levels*

Case Study 1 (CS1) [40] is based on a project for which Logica has developed the front-end software. The outcome of this project is an embedded traffic control system.

Case Study 2 (CS2) is based on a project that Logica has been developing over a period of eight years. Though the project is still in active development, it is already in use by the client. There are new releases of this software system every year. The project consists of the development of one main application together with a couple of small utilities.

3.2.1 Data Collection

The data collected for the two case studies represent defects that were detected and fixed during the testing phase and the post-release use of the projects. The main part of the data collection step was to collect relevant and useful data. To do this, we study the projects' documentation: mainly design documents, UML diagrams, user manuals, test documents. These documents provided insights about: the development of the projects, the defect tracking systems, the severity levels used and how to extract the required details about the defects. Hence, for CS1, we extracted a representative sample of 33 defects based on our knowledge of the project and the recommendations of the designers, the developers and the test analyst working on the project. For CS2, we extracted a sample of 47 defects with the help of the software architect and the developers working on the project. The two subsets were selected to include defects from each severity level used in the two projects. Their number of defects we selected was limited due to the manual effort involved in their selection (together with time constraints). Table 3.3 presents details about the number of fixed defects according to the project's severity levels for CS1. The last column of the table shows the distribution of the selected defects according to the severities from the project. Table 3.4 presents details about the number of fixed defects according to the project's severity levels for CS2 (both the total and the number of defects selected for our case study).

Interviews were conducted with the people working on the projects to get detailed information about the selected defects. These interviews were used to verify that the sample of defects we selected are representative of all defects fixed in the latest versions of the projects.

3.2.2 Data Analysis and Conversion

The information about the 80 defects that we collected (33 defects from CS1 and 47 from CS2) includes the following: the severity levels of the defects, the causes for the defects, the types of the defects, the reasons for assigning a specific severity level to a defect and the ways through which the defects were found. Since this information is project-specific, the IEEE standard in [25] was used to convert the project-specific information about the defects into the project independent attributes, and their values

Table 3.3: *Number of fixed defects according to the severity levels from project CS1*

Severity Level	*Number of Fixed Defects*	
	In the project (1)	Selected for Case Study 1
Showstopper	1	1
Severe	10	10
Medium	93	17
Minor	12	5
Total	116	33

Table 3.4: *Number of fixed defects according to the severity levels from project CS2*

Severity Level	*Number of Fixed Defects*	
	In the project (2)	Selected for Case Study 2
Block	1	1
Crash	11	11
Major	10	10
Minor	123	25
Total	155	47

defined in this standard. As explained earlier, the used attributes are the following: Severity, Effect, Type, Insertion activity and Detection activity.

As can be seen from Table 3.3 and 3.4, the projects used for CS1 and CS2 have four severity levels. However, the ontology uses the severity levels from the IEEE standard [25], which provides five severity levels. Therefore, we defined mapping relation that matches the two sets of severity levels. This relation is shown in Table 3.5.

3.2.3 Data Classification

The data classification step begins with entering the (converted) data from the selected defects into the ontology. Defects are modeled as subclasses. The converted data about the defects was input in the ontology using the Protégé-OWL ontology editor. The input consists of the data about concerning the values of the attributes Effect (which represent quality properties), Type, Insertion activity and Detection activity. The data classification step ends with the automatic classification of the defects into the predefined severity levels (using the rules defined in Table 3.2. The Pellet reasoner applies the classification rules to all (new) classes in the ontology. The result of this is the classification of all defects from CS1 and CS2 input in the ontology into the five severity levels. Upon successful completion of the classification process, the predicted results are displayed in the ontology editor.

Table 3.5: *The relation between the severity levels from the IEEE Standard [25], CS1 and CS2*

Severity Levels		
IEEE Standard [25] and used in the ontology	From the project used in Case Study 1	From the project used in Case Study 2
Blocking	Showstopper	Block
Critical	Severe	Crash
Major	Medium	Major
Minor	Minor	Minor
Inconsequential	Minor	Minor

3.2.4 Results and Comparison

Once the predicted results were ready, we compared them with the results from the manual (original) classifications (after applying the relation in Table 3.5. To summarize the achieved results from both case studies, we created the confusion matrix in Table 3.6. It compares the original classification of the defects from CS1 and CS2 with the automatic (ontology) classification of the same defects. The numbers are given in bold (on the diagonal) represent the number of defects classified into the same severity levels by both classifications. The numbers shown above the diagonal represent the number of defects classified into lower severity levels by the ontology than by the original classifications. The remaining numbers (shown below the diagonal) represent the number of defects classified into higher severity levels by the ontology than by the original classifications. From Table 3.6, we can calculate that the ontology classified 53% of the defects into the same severity levels as originally. 21% of the defects into lower, and 26% into higher severity levels than the original classifications. These results are summarized Figure 3.4. There are two reasons for the differences in the classification results. First, the ontology classification takes into account the point of view of the

Table 3.6: *Summary of the results from the comparison using a confusion matrix (CS1 and CS2)*

		Automatic (Ontology) Classification for CS1 and CS2				
	Severity Levels	Blocking	Critical	Major	Minor	Inconsequential
Manual (Original) Classifications from CS1 and CS2	Blocking	**2**	0	0	0	0
	Critical	0	**16**	5	0	0
	Major	0	11	**12**	2	2
	Minor	0	0	10	**9**	8
	Inconse-quential	0	0	0	0	**3**

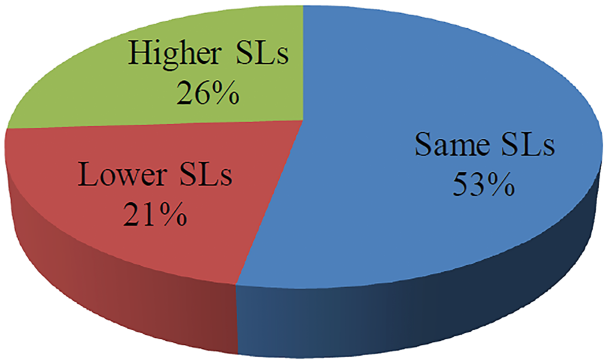

Figure 3.4: *Percentages of the 80 defects classified into the same severity levels (SLs), lower SLs and higher SLs by the ontology compared with the original classifications from CS1 and CS2*

user of the software while preserving the developer's point of view when considering which defects are important for fixing and which defects are important for fixing and which are not. For example, some defects related to the design of and the requirements for the software are classified into higher severity levels by the ontology than originally (other factors also play a role in the classification). This way, these defects will be given a greater chance of being fixed for the next release, which will satisfy more users of the software product.

The other reason is that there are defects assigned the default severity level by the people working on the projects without paying much attention whether this is the correct severity level or not. The default severity level for the project in CS1 is a major while for the project in CS2 it is minor. Since the developed method classifies all defects, the defects originally assigned the default severity level are assigned critical-, major-, minor- or inconsequential severity level by the ontology. Hence, each defect is assigned a specific severity level, and no default severity levels are used. Upon the successful completion of the case studies, we continued with the next step. Since we used the data from these case studies as the training data for MAPDESO, the method had to be tested. The next section presents the validation case study and the data from it served as the test data for MAPDESO.

3.3 Validation

We emphasize that for validation, the already developed ontology and rules were tested on new data from a different project. Similarly to the two case studies from this

Table 3.7: *Number of fixed defects according to the severity levels from the project in VCS*

Severity Levels	*Number of Fixed Defects*			
	In the VCD DB	% of total in DB	Selected for sample of VCS project	% of selection
Top	32	3%	2	4%
High	180	15%	9	18%
Medium	328	28%	16	32%
Low	623	54%	23	46%
Total	1163	100%	50	100%

section, the validation was also conducted in an industrial environment, namely at Logica[2]. It consists of a Validation Case Study (VCS). VCS is based on a project whose development is completed. Currently, VCS is in production use, and Logica provides its maintenance. The project represents an application that handles complex messages between multiple parties. For working on VCS, the approach mentioned in Section 3.2 was used. An important difference of this approach compared with the one in Section 3.2 is that in VCS the severity levels of the selected defects were excluded from the defect reports.

3.3.1 Approach - VCS

The collected data represent fixed defects detected not only from the testing phase of the project but also during its maintenance. A main concern here was to get relevant and useful data. Since we wanted to be as objective as possible when working on VCS, we did not spend any time studying the project's documentation. Instead, for selecting the defects, we relied solely on the help and the recommendations of the project's service coordinator. He provided us with a database containing the defect reports for 1163 fixed defects, which have been detected through testing activities and maintenance in 2011. Applying the method to all of these defects would have taken us much more time than we had for completing the validation. Thus, we considered selecting a sample of 50 defects and this subset includes defects from each and every severity level from the project. Table 3.7 presents the distribution of the 1163 defects and our sample of 50 defects, according to the project's severity levels. It is straightforward to calculate from the table that the distribution of the 50 defects is relatively the same as the distribution of the 1163 defects (in terms of percentages), according to the project severities.

According to the received database and as expected, the defect reports followed project-specific conventions. Therefore, we asked a software engineer working on the project to convert the project-specific information about the defects into the attributes

[2]Now part of CGI, see CGI.com

Table 3.8: *The relation between the severity levels from the IEEE Standard [25] and VCS*

Severity Levels	
IEEE Standard [25] and used in the ontology	From the project used for the validation of the method
Blocking	Top
Critical	High
Major	Medium
Minor	Low
Inconsequential	Low

Table 3.9: *Summary of the results from the comparison using a confusion matrix (VCS)*

		Automatic (Ontology) Classification for VCS				
	Severity Levels	Blocking	Critical	Major	Minor	Inconsequential
Manual (Original) Classifi-cation from VCS	Blocking	**2**	0	0	0	0
	Critical	0	**8**	1	0	0
	Major	0	5	**9**	2	0
	Minor	0	1	8	**13**	1
	Inconse-quential	0	0	0	0	**0**

and their values defined in the IEEE standard [25]. We see in Table 3.7 that the project uses four severity levels. However, the ontology uses the five severity levels from the IEEE standard [25]. Thus, we defined a relation that matches these two sets of severities. Table 3.8 presents the relation. Once ready, we continued with the data classification step. First, we input the defects in the ontology by creating classes for the 50 defects. After that, the converted information about these defects was input in the ontology. In the end, the Pellet reasoner automatically classified all defects into the five severity levels. The predicted results were displayed in the ontology editor.

3.3.2 Results and Comparison

After the predicted severity levels of the selected defects had been present, we compared them with the severity levels of the original classification after applying the relation in Table 3.8. A summary of the results from the comparison between the two classifications is presented in Table 3.9 using a confusion matrix. The numbers on the diagonal (given in bold) represent the number of defects classified into the same severity levels by both classifications.

The numbers shown above the diagonal represent the number of defects classified into lower severity levels by the ontology than by the original classification. The

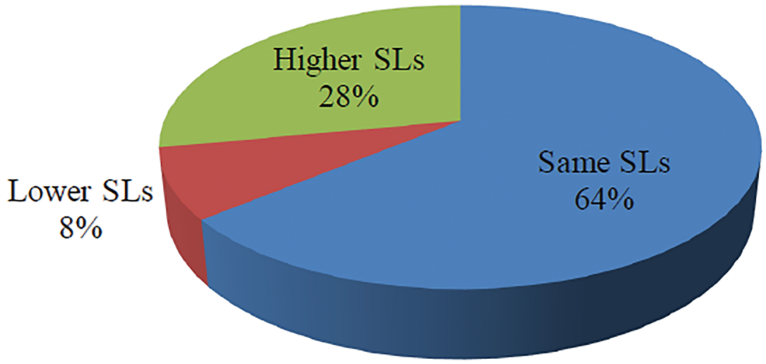

Figure 3.5: *Percentages of the 50 defects classified into the same severity levels (SLs), lower SLs and higher SLs by the ontology compared with the original classification from VCS*

remaining numbers (shown below the diagonal) represent the number of defects classified into higher severity levels by the ontology than by the original classification. We have calculated that the ontology predicted 64% of the defects as having the same severity levels as originally. 8% of the defects as having lower severity levels than in the manual classification from the project, and 28% of the defects as having higher severity levels than in the manual (original) classification. These results are visualized in Figure 3.5.

The reasons for the differences in the classification results are similar to the ones mentioned in the case studies since the same rules are used for classifying the defects from CS1, CS2 and VCS. Hence, we are not going to repeat these reasons again. However, we point out that after comparing Figure 3.4 with Figure 3.5 we notice that the results from VCS are better than the results from the training cases – CS1 and CS2. This can be contributed to the fact that we have dealt with very reliable defect and severity data. Also, to confirm the above observation and, hence, the successful completion of VCS, we present the validation of the results in the next subsection.

3.3.3 Validation of the Results

Validating the above results included presenting them to the software engineer and the service coordinator mentioned earlier. They have highlighted that MAPDESO has performed surprisingly well compared with the original classification from the project. Moreover, they consider that it is very practical that our method uses an IEEE standard [25] for the defects' attributes and their values. The software professionals pointed

out that the method could be very useful for classifying many defects automatically and, then, focus on the defects predicted as having severity level critical and above, for example. In the end, they emphasized that MAPDESO yields very promising results and that they accept these results and find the results acceptable for use in practice.

Following the industrial validation, we decided to compare the performance of MAPDESO with the performances of existing algorithms for data mining tasks and explore which one performs better and why. For the comparison, we used algorithms from the Weka data mining software. The details and the results of the comparison are presented in the next Section.

3.3.4 Comparison

This section presents the comparison of the performance of MAPDESO with the performances of algorithms from the Weka data mining software. However, to compare two entities they have to be measured by a common standard. Therefore, the performances of the automated prediction method and the Weka algorithms have to be compared on the same datasets. As mentioned before, the data from CS1 and CS2 were used during the development of the ontology (as if they were training data). The data from VCS are used for the validation of the method (or, in other words, testing how well it performs). Hence, to have a common standard for the comparison, the data from CS1 and CS2 will be used for training the learning algorithms (called classifiers) from the Weka software. The data from VCS will be used for testing them. Moreover, to conclude the performance, we compared both against the performance of the manual classification from the project used for the validation of the method.

3.3.4.1 Predicting severities of defects using Weka classifiers

The Weka[3] workbench is a collection of state-of-the-art machine learning algorithms and data preprocessing tools [41]. It is designed in such a way that these algorithms can be directly applied to new datasets in flexible ways, which will be very useful for the comparison. Moreover, it provides extensive support for the process of experimental data mining, including preparing the input data, evaluating learning schemes statistically, and visualizing the input data and the results of learning [41]. Weka is used for predicting severity levels of defects. This prediction process consists of the following steps: selecting the classifiers and classifying the test data using Weka. However, these steps are outside the focus of this paper. Therefore, we will present here only the end results.

We selected six classifiers whose performances we compared with the performance of MAPDESO. These classifiers are the following: ZeroR, DecisionStump, NaiveBayes, IBk (k = 5), SimpleLogistic and SMO [41]. We selected exactly them as they are common

[3] http://www.cs.waikato.ac.nz/ml/weka/

in the data mining field. Then, these classifiers were trained on the data from CS1 and CS2 and tested on the data from VCS. All results were displayed in Weka. We decided to use precision, recall and F-measure for the comparison of the performances. We chose these statistics because they provided useful and relevant to our research information for the performances of the classifiers, as evident by their definitions and mentioned in [42]. The statistics are defined below using the definitions provided in [42].

Precision represents the number of correct results divided by the number of all returned results. The Recall is the number of correct results divided by the number of results that should have been returned. *F-measure* is the weighted average or harmonic mean of precision and recall. It can be interpreted as a weighted average of precision and recall. It is calculated using equation (3.1) below. Precision, recall and F-measure reach their worst values at 0 and their best values at 1.

$$\mathrm{F-measure} = \frac{2 * \mathrm{precision} * \mathrm{recall}}{\mathrm{precision} + \mathrm{recall}} \tag{3.1}$$

A high recall allows engineers to operate on the returned results without revisiting and reassessing the complete defects. However, a recall of 1 at the expense of a very low precision (e.g. smaller than 0.5) would not yield a good speedup. An engineer would be confronted with a list that is almost as long as the list of all defects that he or she needs to sort out. Therefore, a good algorithm for the problem discussed in this paper needs to balance recall and precision and F-measure is a good metric to compare algorithms.

Next, we present the results from classifying the test data using the six classifiers. Table 3.10 shows the results using the percentage of defects classified correctly together with the precision, recall and F-measure per classifier per severity level. The table also contains the weighted average values (abbreviated to W. Avg.) of the four statistics for each classifier. The results from Table 3.10 are visualized in three figures. For precision per severity level for the classifiers are shown in Figure 3.6. In the same way, Figure 3.7 presents the results for recall and Figure 3.8 visualizes the results for F-measure. These figures also contain the weighted average values (abbreviated to W. Avg.) of the three statistics for each classifier.

3.3.4.2 Comparison of the performances

The comparison of the performances starts with presenting the results from testing MAPDESO using the same three statistics as above. As mentioned earlier, the performance of MAPDESO was tested using the data from VCS during the validation process (see Section 3.3). The results of the testing are given in Figures 3.6, 3.7 and 3.8 together with the results for the six classifiers.

For an easy and straightforward comparison of the performances of the chosen classifiers with the performance of MAPDESO, we use the weighted average values of

Table 3.10: *The results from classifying the test data (VCS data) by the six chosen classifiers and by MAPDESO*

Classifier	Severity levels	Precision	Recall	F-measure
Classifier 1: ZeroR	Blocking	0.00	0.00	0.00
	Critical	0.00	0.00	0.00
	Major	0.32	1.00	0.49
	Minor	0.00	0.00	0.00
	Inconsequential	0.00	0.00	0.00
	W. Avg.	0.10	0.32	0.16
Classifier 2: Decision Stump	Blocking	0.00	0.00	0.00
	Critical	0.25	0.56	0.35
	Major	0.00	0.00	0.00
	Minor	0.63	0.83	0.72
	Inconsequential	0.00	0.00	0.00
	W. Avg.	0.34	0.48	0.39
Classifier 3: NaiveBayes	Blocking	0.00	0.00	0.00
	Critical	0.36	0.44	0.4
	Major	0.4	0.5	0.44
	Minor	0.68	0.57	0.62
	Inconsequential	0.00	0.00	0.00
	W. Avg.	0.51	0.5	0.50
Classifier 4: IBk with k = 5	Blocking	0.00	0.00	0.00
	Critical	0.56	0.56	0.56
	Major	0.5	0.38	0.43
	Minor	0.62	0.78	0.69
	Inconsequential	0.00	0.00	0.00
	W. Avg.	0.55	0.58	0.56
Clasifier 5: SimpleLogistic	Blocking	0.00	0.00	0.00
	Critical	0.35	0.89	0.50
	Major	1.00	0.13	0.22
	Minor	0.76	0.83	0.79
	Inconsequential	0.00	0.00	0.00
	W. Avg.	0.73	0.58	0.52
Classifier 6: SMO	Blocking	1.00	0.5	0.67
	Critical	0.46	0.56	0.50
	Major	1.00	0.13	0.22
	Minor	0.76	0.83	0.79
	Inconsequential	0.00	0.00	0.00
	W. Avg.	0.56	0.58	0.52
MAPDESO-automated prediction method	Blocking	1.00	1.00	1.00
	Critical	0.57	0.89	0.7
	Major	0.5	0.56	0.53
	Minor	0.87	0.57	0.68
	Inconsequential	0.00	0.00	0.00
	W. Avg.	0.70	0.64	0.65

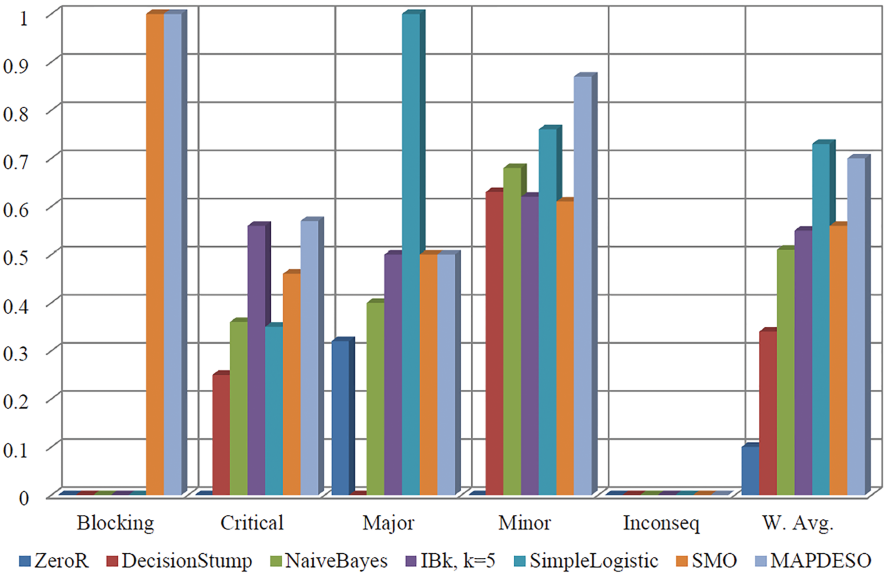

Figure 3.6: *The results for precision per severity level for the six classifiers and for MAPDESO*

Table 3.11: *Summary of the comparison between the six classifiers and MAPDESO*

Classifiers (classification methods)	Weighted average values of		
	Precision	Recall	F-measure
ZeroR	0.10	0.32	0.16
DecisionStump	0.34	0.48	0.39
NaiveBayes	0.51	0.5	0.5
IBk with k = 5	0.55	0.58	0.56
SimpleLogistic	0.73	0.58	0.52
SMO	0.56	0.58	0.5
MAPDESO	0.70	0.64	0.65

the three statistics. Also to Figures 3.6, 3.7 and 3.8 we created Table 3.11. It contains the weighted average values of the three statistics for the classifiers and the method. It is visible that MAPDESO has the second highest precision (3.6, Table 3.11), the highest recall (Figure 3.7, Table 3.11) and the highest F-measure (Figure 3.8, Table 3.11). Only the SimpleLogistic classifier has a better precision than that of our method (Figure 3.6, Table 3.11) and the reasons for this are explained below. Figure 3.9 shows the visualization of Table 3.11. First, we have to look at the specific precision values for the different severity levels for the SimpleLogistic classifier (see Figure 3.6).

It is easy to notice that the precision for SimpleLogistic is 1 for severity level major.

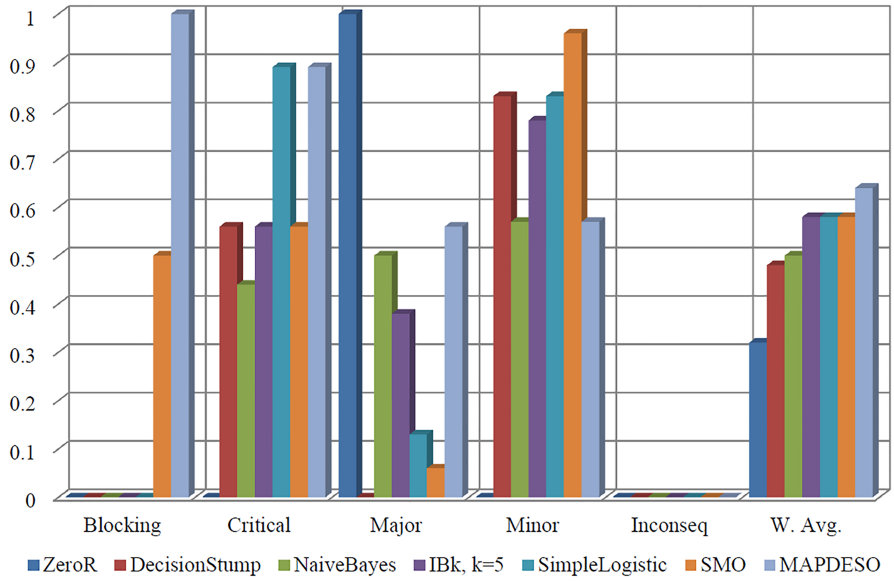

Figure 3.7: *The results for recall per severity level for the six classifiers and for MAPDESO*

With such a high precision, it is obvious that the weighted average precision for this classifier will be high, as well. However, if we look at this classifier's recall for severity level major, we see that it is only 0.13. Although this classifier returns correct results only for severity level major (precision is 1), it returns a very small portion of the correct results. These results should have been returned to the aforementioned major severity level (recall is 0.13). In other words, the returned results are very exact but very far from complete. On the other hand, the automated prediction method has a precision of 0.50 and a recall of 0.56 for severity level major. This means that although the method returns correct results one-half of the time for severity level major (the precision is 0.50), it returns more than half of the correct results that should have been returned to this severity level (the recall is 0.56). So, the returned results are correct one-half of the time and complete more than half of the time. Moreover, if we look at the weighted average F-measure for SimpleLogistic, we notice that it is 0.52. This is lower than the weighted average F-measure for the automated prediction method (0.65 as given in Figure 3.9) despite the fact that SimpleLogistic has a precision greater than that of the method. Therefore, based on the above explanations, it is safe to say that the overall performance of the SimpleLogistic classifier is not as good as the performance of MAPDESO when classifying defects into all severity levels.

We can apply similar reasoning to the other five classifiers when comparing their overall performances with the performance of the developed method when classifying defects into all severity levels. At a specific severity level, Figures 3.6 and 3.7 show that

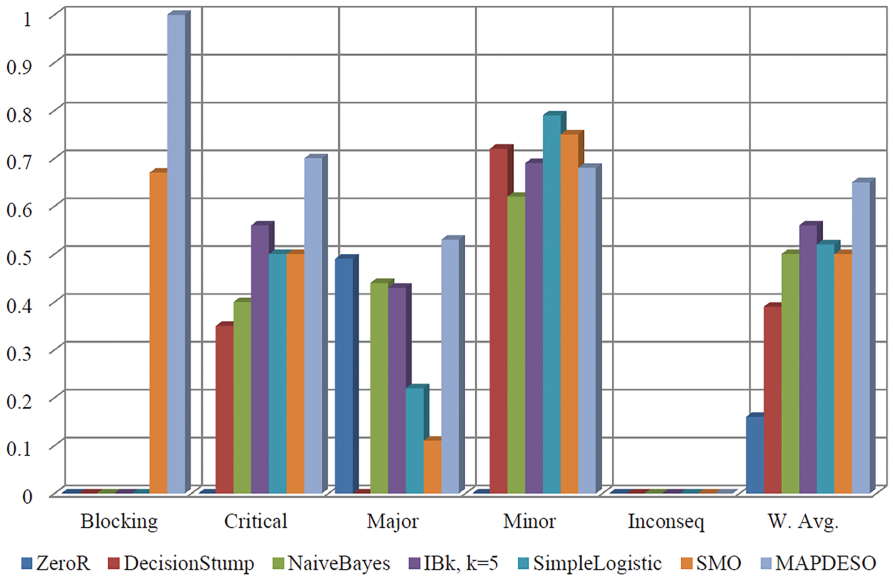

Figure 3.8: *The results for F-measure per severity level for the six classifiers and for MAPDESO*

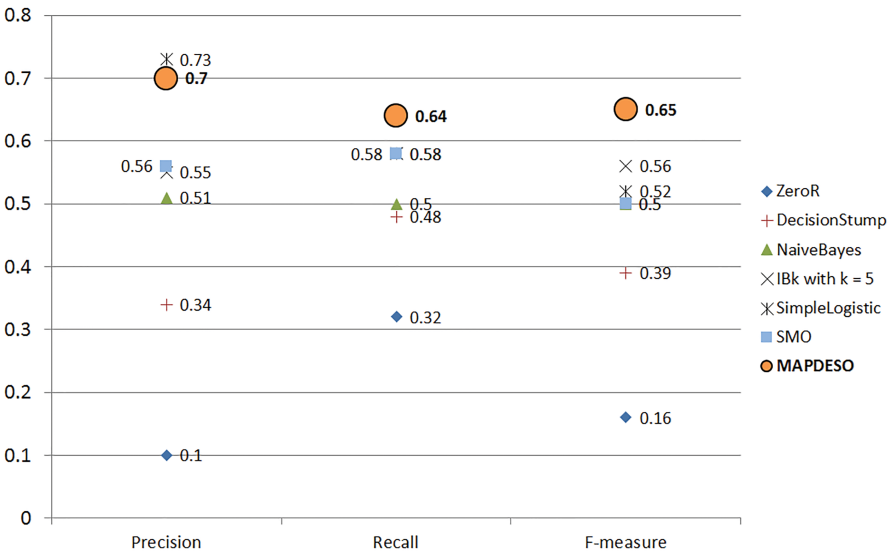

Figure 3.9: *Summary of the comparison between the six classifiers and MAPDESO*

one or more classifiers might have a precision and/or a recall greater than or equal to those of the automated prediction method. For the other severity levels, the method has greater values of precision and recall. Based on Figures 3.6 - 3.9, we conclude that the overall performance of MAPDESO is better than the performances of the chosen classifiers when classifying defects into all severity levels. More importantly, we see in Figure 3.8 that the automated prediction method has F-measure of 1 and 0.70 for severity levels blocking and critical, respectively. These are by far the best F-measure values compared with the respective values of the six classifiers. Hence, the method performs the best compared to the performances of the classifiers when predicting which defects will be assigned the most important severities, namely blocking and critical.

3.4 Related Work

In our approach, we follow the IEEE Standard Classification for Software Anomalies (IEEE Std 1044™ 2009) [25] which provides a uniform framework for the attributes of the defects and their terminology. In addition to this, we use a technique for the field of Artificial Intelligence (AI) techniques, namely ontologies. Ontologies provide a generic framework for modeling and reasoning about knowledge in a particular domain. We will use ontologies for automatic classification to predict the severity levels of defects automatically. Different projects define and use different sets of severity levels. So following a uniform framework will be valuable for providing a single set of severity levels that is known to everybody – software architects, developers, test analysts. Using a standard for defect severities within on company reduces the time, and cost for retraining people, when they switch projects and reduces severity classification mistakes. The IEEE Standard [25] provides a uniform approach to the classification of software anomalies. It contains a classification of defects, which defines a core set of widely applicable classification attributes. Sample values for the most common attributes are provided together with definitions and examples for both the attributes and their values. For our research, the following attributes were selected from the standard: Severity, Effect, Type, Insertion activity and Detection activity. The idea of using specifically these attributes is to provide a uniform framework for the attributes of the defects and their values so that the method can be used across multiple software projects and systems.

Different techniques have been researched for predicting severity levels of defect reports [43], as well as for predicting the presence or absence of faults [23] and defects [44]. These techniques include standard text mining methods, logistic regression and machine learning techniques, and the Six Sigma methodology. Though they have proven to be very useful, they base their results on syntactical text mining analysis and statistical methods. For our research goal, we use ontologies for an automated prediction process, which is uses richer semantical concepts of the impact defects have

on the quality properties of the software such as functionality, usability, security, ... etc.

The severity levels assigned to defects are used to find out what is the impact of that defect on the deployment of the software. Hence, an important aspect of this is the reason why a specific defect is assigned one severity and not another. This will help both the developers of the software product and its users to agree to the assignment of the severity levels. Our ontology-based approach will be better able to provide an explanation of why it assigns a severity level, than statistical techniques are.

Menzies and Marcus present a new and automated method, which assists the test engineer in assigning severity levels to defect reports [43]. They have named the method SEVERIS (SEVERity ISsue assessment) and it is based on standard text mining and machine learning techniques applied to existing sets of defect reports. The tool is designed to automatically review issue reports and trigger an alert when a proposed severity is anomalous. Moreover, the paper presents a case study on using SEVERIS with data from NASA's Project and Issue Tracking System (PITS). The case study results indicate that SEVERIS is a good predictor for issue severity levels, while it is easy to use and efficient. The idea behind our research is similar to the study in [43] – an automated method for predicting what severity levels should be assigned to defects. However, we base our method on the software development process and software quality properties to decide what severity level should be assigned to a defect.

Zhou and Leung investigate in [23] the accuracy of the fault-proneness predictions of six widely used object-oriented design metrics with particular focus on how accurately they predict faults when taking fault severity into account. Their results indicate that most of these design metrics are statistically related to fault proneness of classes across fault severity and that the prediction capabilities of the investigated metrics greatly depend on the severity of faults. This work is similar to the one in [43] in the sense that the authors use logistic regression and machine learning methods for their empirical investigation.

In our research, we focus on predicting the severity levels of defects using ontologies and automatic classification. This is achieved by developing an ontology and classifying the defects using developed classification rules. Additional motivation for the current work comes from the research conducted by Suffian [44] who establishes a defect prediction model for the testing phase using the Six Sigma methodology. The author's aim is to achieve zero-known post-release defects of the software delivered to the end users. This is done by identifying the customer needs through the requirements for the prediction model. The author states that his work focuses on predicting the total number of defects regardless of their severity, or the duration of the testing activities. Also future effort can focus on improving the defect prediction model to predict defect severity in the testing phase. Therefore, we aim at predicting the severity levels of defects that have been found during testing at the system level (though we also consider defects found during coding and maintenance).

Another area of related research is research aimed at the combination of ontologies

and software design, which emphasizes error detection [45][46]. This research proves to be very useful because it enhances the software design quality, as stated by Hoss [45]. It also improves the practice in ontology use and identifies areas to which ontologies could be beneficial other than, for example, knowledge sharing, and reuse, as explained by Kalfoglou [46]. In our work, we combine ontologies with knowledge of the software development process for automatically predicting the severity levels of defects (which have already been detected and reported). Thus, our goal is different from the ones mentioned in [45] and [46] though the means to achieve it are similar to some extent.

3.5 Threats to Validity

In this section we discuss threats to validity.

3.5.1 Conclusion Validity

We use different projects, and we extracted defects that reflect the nature of the whole set of defects. We did this with the help of designers and developers that work at that company. There are many machines learning algorithms, and each one has many different parameters. We compare MAPDESO with six machine learning algorithms using WEKA, and we use the default parameter setting in WEKA.

3.5.2 Internal Validity

We ensure that the selected defects from the projects' defect reports represent the natural defect reports. We extracted the sample of defects with help and recommendation of professionals that worked at the company, such as designers, developers, and a test analyst.

3.5.3 Construct Validity

All the documentation of the projects is in Dutch. We managed to translate defect reports into English. Then we convert the reports into IEEE standard classification for software anomalies. We validated our English reports and the conversions to IEEE standard with professionals who work at the company.

3.6 Conclusions

In this work, we have presented MAPDESO – a Method for Automated Prediction of DEfect Severity using Ontologies. It considers the quality properties affected by defects, the types of the defects, the insertion activities and the detection activities of the defects. This way, it takes into consideration the point of view of the user of the software

system while preserving the developer's point of view when predicting the severity levels of defects. This method uses defect attributes and their values from the IEEE Standard Classification for Software Anomalies [25] to create a uniform framework for reporting the defects and making it applicable to various software projects. Last but not least, the method uses AI techniques – ontologies and ontology reasoning, to automatically predict the severity levels of the defects input in the ontology according to the developed classification rules for the ontology. The chapter started with an introduction to the problems we are solving with the developed method and the work related to our research. After that, we provided information about ontologies, ontology development, and languages. We continued with presenting the details of MAPDESO. Next, the case studies used for the development of the ontology were described. Then, the automated prediction method was validated using a validation case study. In the end, the method's performance was compared to the performances of six well-known classifiers from the Weka machine learning workbench. The results from the comparison led to the conclusion that MAPDESO performs better than the chosen classifiers. Based on the results from the validation process and the comparison process, we state the following:

- The automated prediction method performs well compared to the manual (original) classifications of the defects obtained from the conducted case studies. It uses as few as four attributes from the standard to predict a fifth attribute – the severity levels.

- The method is very practical because it uses an IEEE standard [25] for the defects' attributes and their values. Hence, if future projects adopt it, they will have a standardized framework for the defects' attributes. This implies that people will be able to move from project to project, if needed, without wasting extra time for retraining.

- It yields very promising results that can be useful for medium-to-large projects with many defects.

- MAPDESO outperforms the chosen Weka classifiers, and the performance of the method reaches its peak when predicting which defects will be assigned the most important severity levels – blocking and critical.

Last but not least, MAPDESO predicts the severity levels of defects detected from system-level testing, coding, and maintenance. However, it is worth mentioning that MAPDESO could be adjusted so that it can be used to predict the severity levels of defects detected from any phase of the software development process.

3.7 Future Work

Future work will be aimed at further automating the prediction method. This could be achieved by automating the conversion of defect reports into the standard representation. Completing such a step would require natural language processing, data mining algorithms and automated reasoning about designs. We would also like to increase the level of automation of reasoning by focusing on defect propagation that links defects found at unit-level to use cases at the system level. In this situation, the severity prediction will be based on the impact found via defect propagation and the importance of the use cases that are impacted in the application domain.

Future work would also be aimed at applying MAPDESO in practice. This could be achieved by implementing it in a defect tracking system. This implementation could be either as the sole method for predicting the severity levels of defects, or as a method providing severity levels as suggestions to be confirmed by software engineers or clients. A possible continuation of this work is to apply the automated prediction method to other projects, for example, open-source projects. Lastly, we would also like to try blending machine learning with our method to get the best of both.

Chapter 4

Establishing an Infrastructure for Empirical Research on UML Diagrams

*In this chapter, we present a solution for collecting UML diagrams in image formats. We present our developed crawler that collects UML diagrams in image formats from the Internet. Then we illustrate our classifier that classifies UML diagrams in image formats from other images, such as screenshots and natural pictures. Finally, we demonstrate our **Img2UML** tool that converts UML models in image formats into XMI files.*

This chapter is based on the following publications:

- Bilal Karasneh, Michel R. V. Chaudron. **Extracting UML models from images.** *In Proceedings of the 5th International Conference on Computer Science and Information Technology (CSIT2013), pages 169-178, Amman, Jordan. 2013.*
- Bilal Karasneh, Michel R. V. Chaudron. **Img2UML: A System for Extracting UML Models from Images.** *In Proceedings of the 39th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2013), pages 134-137, Santander, Spain. 2013.*
- Truong Ho-Quang, Michel R. V. Chaudron, Ingimar Samúelsson, Jóel Hjaltason, Bilal Karasneh, Hafeez Osman. **Automatic Classification of UML Class Diagrams from Images.** *In Proceedings of the 21st Asia-Pacific Software Engineering Conference(APSEC2014), pages 399-406, Jeju, Korea. 2014.*

Companies have a huge amount of information at their disposal, but this information is available in poor formats, like paper documents, or in poorly structured electronic formats, such as images, PDF or DOC. These companies have a need to convert this important information into richer formats that can be easily searched and modified [47]. In software engineering, this challenge becomes bigger because software documentation is rich in graphical content. These graphics or images mostly are models, charts, schemes, etc. The computational challenge here is a lack of mapping from a pixel-based diagram to the underlying engineering model conveyed by the diagram [48]. UML is used for modeling software because it can show a high-level description of a system. UML models are created during different stages of the development process and also during maintenance. UML models are a graphical representation, and they are mostly available as images on the internet and as part of software documentation such as software architecture and software documents. So we recognize the needs for extracting UML models from images. We summarize two reasons but not limited to:

First, in software projects, UML models are typically made during the early (design) phases of a project. As projects progress, emphasis in developer activity shifts to coding and tend to ignore updating the UML model. One reason for the lack of updating UML models is that the UML models have been copied from a CASE tool and pasted into a software design document, which is created using a text processing tool. In this way, design is stored together with explanatory text. In such word-processing tool, the UML model is now represented in an image format, which is not editable. Often, the software design document is used in subsequent development, but the CASE-tool version of the model is neglected, leaving a project only with an image format version of their design. Clearly it is desirable to recover the UML model from such image formats for updating and maintenance.

Second, in academia, extracting UML models information from image formats is very useful for student, because it allows them to reuse models. Many UML models are available on the internet, but for reusing these models, they need to be re-drawn, which is annoying and wasting time and effort. Next, we present our developed crawler for collecting UML models from the internet.

4.1 Overview of UML Crawler

We know that it is convenient to use internet search engines. Google, as the biggest search engine in the world, has information of billions of websites and keeps them up to date. The search engine is the technology that started in 1995. The search engine collects information from the Internet, classifies them and provides the search function of the information for the users. It has become one of the most important Internet services. A web crawler is a program that collects information from the Internet automatically.

The process of fetching data works the same as our online surfing. The web crawler mostly deals with URLs. It gets the content of the files according to the URL and classifies it. Thus, it is very important for the web crawler to understand URLs. When we want to read a web page, the web browser, sends a request to get the source of the page. Next, the browser interprets the content and shows it to us. The address of a web page (URL) is a special type of universal resource identifier (URI). URI is the identifier for every resource on the website: HTML documentation, images, videos, programs and so on. It contains three parts: (1) the protocol, (2) the IP address of the computer node where the resource is, (3) and the path of the resource (file) at the computer node.

4.1.1 Methodology of making UML Crawler

Our methodology is:

1. Create a crawler for collecting UML models from the internet.
2. Create a tool that can recognize models information from the images (such as class names and relationships).
3. Transform the extracted model information into an XMI file.
4. Store images and XMI files in a database.

We develop our developed crawler called (*UMLCrawler*) that can collect a large number of UML diagram images from the Internet. The crawler can download images of UML efficiently and store information about these images in a database.

4.1.2 Differences with other solutions

Even using the large search engines like Google and Yahoo, it is difficult to get a large number of UML diagrams in a short time. No downloading service is available to save the results of a searched query to the local disk. Also, there is noise - false positive to the search. There are several tools that are specially designed for downloading images from the Internet. However, they have drawbacks. Some of the tools cannot meet the requirement of downloading images with a large number and full sized. We show some examples:

- Google has provided an API for downloading images from the result of image search [49]. However, the API has a limitation of getting a maximum of 64 results on eight pages. It is far from enough.
- Firefox browser[1] has a plug-in named “Save images” that can save images from the current tab page the user has opened. The problem is that the results of Google

[1] https://www.mozilla.org/en-US/firefox/new/

image or Yahoo image search is shown on the web page are just thumbnails. The images downloaded are too small to use. Furthermore, the plug-in can only download images from pages that have opened in the browser, which is not possible to open all the pages of the list of images and download them one by one.

- Bulk Image Downloader[2] is software that can download full sized images from almost any thumbnails web gallery. However, it takes commercial software and has the same limitation that we should open the every page in the resulting search to download the images.

Our crawler called *UMLCrawler*, which collects UML models stored in image formats from the Internet and download it on the local drive. The crawler does not have limitations on numbers of downloaded images.

4.1.3 Crawler Requirement

The requirements of our crawler are:

1. We want to collect as much as UML models automatically.
2. We want to finish the process efficiently by avoiding false positive results.

To finish the task we implement a web crawler that collects UML images from the internet. A web crawler with high performance should have two features:

1. It should be able to grab a great quantity of data from the internet.
2. It should run on a distributed system, because the quantity of data is extremely large, and different users can have different results that enrich the collection of UML models downloaded.

We will explain this point in the next subsection.

Different from a normal web crawler that gets every piece of information from the internet, the crawler we want to implement is just to download images of UML models from the internet to establish the database. Thus, we can use the result of an image searching engine such as Google Images and Yahoo Image search, because it would be more efficient and powerful.

4.1.4 Using Google Images

Google is the most widely used search engine in the world. We believe that Google Image is the best fits in our requirements because it can get a large number of images based on searched keywords. When we go to the image search page of Google, we

[2] http://bulkimagedownloader.com/

can get the list of images from various websites based on used keywords. The time for Google image to reply to a search is extremely fast because the information of billions of images has already been saved in the server of Google. Thus, our web crawler can use the results of Google image search as a starting point; the efficiency is great.

In October 2009, Google has released the function to search by images[3]. When an image is uploaded, the algorithm is applied on it to abstract the features from it. The features can be textures, colors, and shapes. Then, the features are sent to the backend of Google and compared with the images in their database. If there is an image that has similar features with the original picture, the algorithm takes it as a confident match. Then, Google will list the confident matches on the website for the user. The advantage of this algorithm is that it simulates the process of human beings looking at an image.

If the image for the query has a unique appearance, the result will be very good. The result for unique landmarks like the Eiffel Tower is fantastic. However, when we are using the function to search for UML models, the result is not so efficient. The reason is that UML class diagram is mainly combined with shapes like rectangles, arrows, and text. There are no unique symbols. When we upload a UML class diagram to the Google search by image, the features Google will extract are not special. Thus, the result is not only UML class diagram but also graphs with curves and rectangles that relate to math research, stock markets, and medical research and so on. The function we want to use is to search images by keywords, because at least for UML class diagrams, the accuracy of searching by image does not take advantage of searching using keywords.

Google provides details of the personal result, which collects images related to the most websites that the user has visited or is interested in, which is helpful for collecting models the user wants. Because of this, we need to run our crawler on a distributed system; that different user can have different results based on their interested and visited websites using Google search. This enriches our collection of UML models. Other search engines also have similar functions of image search, but Google provides a better user interface and more powerful filter for the images. The URL of Google image search can be easily constructed with different parameters. We can use Google image search in our crawler without going to the web browser first.

4.1.5 Implementation of UMLCrawler

The crawler is built using VB.NET. Users can use different keywords as they do on Google Image. Users can mention the number of images they want to download, or a time duration that the crawler can keep downloading. The crawler does not rely on the API of Google Image. The crawler gets images URLs from Google Image, and downloads all images using the URLs. After downloading an image, the information of the image will be saved in the database. This information contains: image URL, image width, and height. The crawler save images URLs to avoid re-download it again

[3]https://support.google.com/websearch/answer/1325808?hl=en

Table 4.1: *Table "Img", contains information about images that are downloaded by the crawler*

Key	Type
ID	Auto-increment
url	Text
Width	Text
Height	Text
image_Name	Text
Comments	Text
isUML	Boolean

Table 4.2: *Table Blacklist, contains blacklist URLs*

Key	Type
ID	Auto-increment
url	Text

in the future by comparing a new URL with URLs in the database. Users can choose the downloading path on the local driver, and select an existing database or a new database to save images path and the other information. Furthermore, the crawler has extra features, such as user blacklist. From the user blacklist, the user can skip downloading images from URLs that do not contain UML models, and images URLs that are not UML images.

4.1.6 Crawler Database

We designed the database structure to keep the information of the downloaded images. Most of the information is saved in one table called "Img". The table “Img” contains images URLs, width, height, comments and isUML. Comments are used to store users’ feedback about the images. Another attribute we have added to the database is "isUML". It is a Boolean value that shows whether the image is a UML model or not. This is a manual feature that users can set when they are exploring the images. Although the keywords can be such as "uml class diagram", no search engine can ensure that all the images that found are strictly UML class diagram. Therefore, there should be a key that captures which images are indeed UML diagram. As not all images downloaded are UML, there is a list needed that can save the URLs of such images to save the trouble to download them again next time. Thus, a "blacklist" is added to the database. From this information, we want to find out which websites can provide UML models more than others. We use MySQL to create the database. Table 4.1 shows the "Img" table. Table 4.2 shows the "Blacklist" table.

The whole process of collecting UML images by the crawler has took an hour. The

time used depends not only on the algorithm but also on the Google image response because they delay some responses because of continued requests from the same user. We have manually established the "Blacklist" during the collection. Images that are not UML class diagrams, too blurred to distinguish or a screen shot that contains only a part of a UML class diagram are put into the "Blacklist". As a result, 1153 images have been put into the "Blacklist" manually.

In the next subsection, we are describing the limitations of the crawler.

4.1.7 Limitations of UMLCrawler

UMLCrawler is based on Google image search; this can be considered as an advantage and disadvantage at the same time. The first weak point is that the pages of the result of images search are limited (50 pages). The average number of images within one page is 20. Thus, the maximum number of images the program can get with one string of keywords is 1000. Because the URL of some images are out of date, so the actual result is a bit less than 1000 images. Therefore, to collect more images we use several keywords for the search to build our database. The second weak point is the accuracy. As keywords motivate the searching process, the images found are those with descriptions that include the keywords. Thus, not all images found are UML class diagrams. Some of them maybe the screenshot of a presentation named "UML class diagram" or a photo of a book that relates to UML class diagram instead of low-resolution images that we are added in the Blacklist.

The percentage of images and URLs added to the Blacklist are high (1153 from 2564 = 45%). After we had investigated the 1153 images in the Blacklist, we found that most of them are not UML class diagram and low-resolution images (82%). Because of this, we identify the need to build a classifier to automatically classify the collected images. In the next section, we describe our classifier.

4.2 UML Image Classifier

To enrich our collection of UML models, and to decrease the time and effort for validation of the collected images whether it is UML models or not, we build a classifier for recognizing UML class diagram images, called (*UMLImgClassifier*). The classifier works by extracting relevant features from images and processing these features with a machine learner. The classifier can distinguish between UML class diagram images and non-UML class diagram images. In the next subsection, we present our approach to classifying UML class diagram in images.

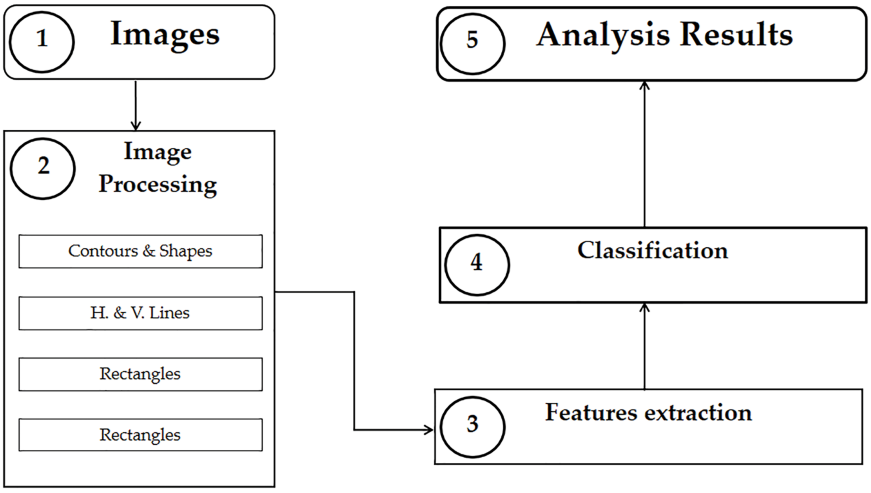

Figure 4.1: *Overall Classification Process*

4.2.1 Classifier Approach

Figure 4.1 shows the overall approach of our classifier. Images are the input. Then the images are processed by some image processing techniques, such as recognition of contours and lines. The output of the recognition process is listed of 23 features. These features are used to build the classifier of UML class diagrams. The classifier was trained with 1300 UML class diagram images collected via *UMLCrawler*, where 50% are UML class diagram and 50% are not. Figure 4.2 shows the steps of image processing that used for extracting UML class diagrams features. From the image processing, we can distinguish some main characteristics of UML Class diagrams, such as rectangles, lines, the number of colors, etc. In the next subsection, we explain the features we extract for solving our classification.

4.2.1.1 Images Features

Three key factors can be used to describe UML class diagrams:

1. Classes, in the form of rectangles.
2. Connections between classes in the form of connecting lines.
3. Rectangles that represent classes are divided into three sections maximum, which are: the class name, the attributes, and the operations.

These defined characteristics can be valid for other types of diagrams and charts such as process diagrams and object diagrams. Therefore, it is important to extract more information from UML class diagrams, which are more specific.

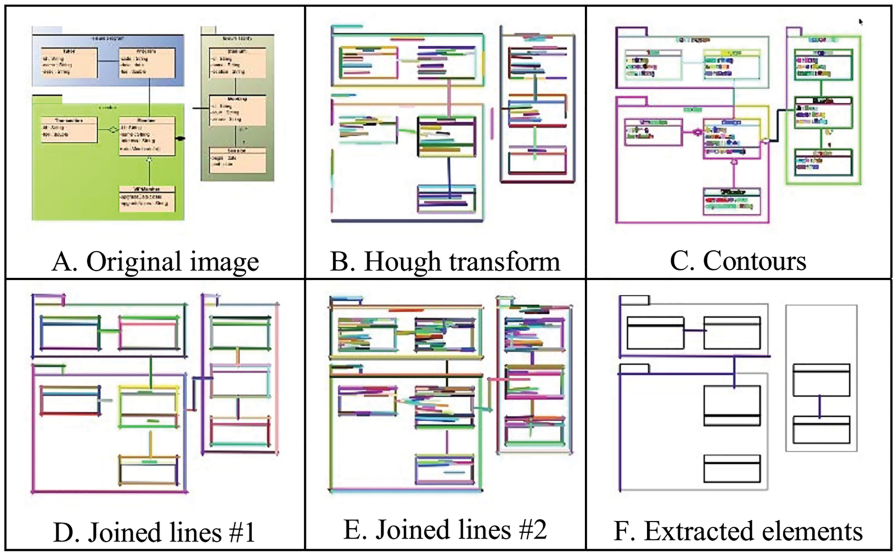

Figure 4.2: *Image processing*

We extracted 23 features, which are calculated by using image processing techniques. Table 4.3 shows the extracted features. In the next subsection, we explain how we use these features to build the classifier.

4.2.1.2 Classification Algorithms

We made an experiment using WEKA because it supports many classification algorithms to find the best classification algorithm based on our extracted features. The classification algorithms we consider are [50]:

1. Decision Table (DT);
2. J48 Decision Tree (J48);
3. Logistic Regression (LR);
4. Random Forest (RF);
5. REP-Tree (RT); and
6. Support Vector Machine (SVM).

We use the information Gain Attribute Evaluator (InfoGain) to find out the influence of extracted features. Then, we apply the Correlation-based feature selection (CFS) algorithm [51] on the extracted features. We prepared several sets of predictors that are used in this evaluation: the top 3, top 6, top 9 and "top-all" of the most suitable

Table 4.3: *Extracted Features*

Feature No.	Name	Description
F.01	Rectangles' portion of image, percentage	Calculated by dividing the sum of the area of all the rectangles with the area of the image itself
F.02	Rectangle size variation, ratio	Calculated by dividing the rectangle size on the standard deviation of the rectangle average size
F.03-06	Rectangle distribution, percentage	The image is divided into four equally sized sections and the area of the rectangles inside the sections are then divided by the total area of the rectangles. The four sections sum up to 100%
F.07	Rectangle connections, percentage	Calculated by counting all rectangles that are connected to at least one rectangle, and dividing that number by the total amount of rectangles in the image
F.08-10	Rectangle dividing lines, percentage	The rectangles are split into three groups, with rectangles that have: no dividing lines (F08); one or two dividing lines (F09); or three or more dividing lines (F10). This produces three numbers that represent the percentage of rectangles within each group
F.11-12	Rectangles horizontally and vertically aligned, ratio	Sides of rectangles, horizontal (F11) and vertical (F12) that are aligned with sides of other rectangles are counted. The numbers are then divided by the number of detected rectangles in the image-resulting in two ratios on rectangle horizontal and vertical alignments
F.13-14	Average horizontal and vertical line size, ratio	Average size of horizontal (F13) and vertical (F14) lines that are larger than ¾ of the images width or height, divided by the images width or height, respectively
F.15	Parent rectangles in parent rectangles, percentage	Rectangles that have rectangles within them can possibly be packages. This feature is the percentage of the area of those parent rectangles that is within other parent rectangles
F.16	Rectangles in rectangles, percentage	This feature is calculated in the same manner as (F15), but with rectangles, instead of a parent rectangles
F.17	Rectangles height and width ratio	The average ratio between the height of the rectangles and the width of the rectangles
F.18	Geometrical shapes' portion of image	The same as F01, but with rhombuses, triangles, and ellipses
F.19	Lines connecting geometrical shapes, ratio	The number of connecting lines from shapes, other than rectangles, divided by the number of detected shapes in the image
F.20	Noise, percentage	F.21-23 Color frequency, percentage Three most frequent colors in the image are found. Then a percentage out of all appearing colors are found for the three colors

features. Then, we use these predictor sets to all classification algorithms to get their false-positive (FP) and true-positive (TP) rates on our dataset.

4.2.2 Experiment Description

In this subsection, we explain the dataset used and explain the results.

4.2.2.1 Dataset

We collected 1300 images[4], 632 are UML class diagrams and 632 non-UML class diagram. The non-UML images include 60 sequence diagrams and 155 charts.

4.2.2.2 Evaluation Measures

We use confusion metrics to evaluate the machine learning classification algorithm. Table 4.4 show the confusion metrics. We use Sensitivity and Specificity to evaluate

Table 4.4: *Confusion Matrix*

Actual Result	Prediction Result	
	Y	N
Y	TP	FN
N	FP	TN

the performance of the classification algorithms. Specificity represents the ability to exclude non-UML CD images, and sensitivity represents the ability to include UML CD images. The two metrics are calculated from the confusion matrix as below:

$$Specificity = TNR \frac{TN}{TN + FP} \tag{4.1}$$

$$Sensitivity = TPR \frac{TP}{TP + FN} \tag{4.2}$$

In our case, the exclusion of non-UML class diagrams is more important than the inclusion of UML class diagrams. As a result, specificity is considered more important than sensitivity. The two measures range from 0% to 100%.

4.2.2.3 Machine Learning Settings

We use 10-fold cross-validation [41] for performance evaluation where all images are randomly split into ten exclusive folds. The default settings suggested from Weka were used for the classification algorithms.

[4]The dataset can be found online via: http://bitly.com/dtsUMLClassifier

Table 4.5: *Result of InfoGain*

No.	Features	InfoGain Value	No.	Features	InfoGain Value
1	F.09	0.473	13	F.18	0.111
2	F.20	0.433	14	F.14	0.086
3	F.01	0.374	15	F.10	0.07
4	F.13	0.352	16	F.21	0.055
5	F.08	0.306	17	F.19	0.052
6	F.02	0.302	18	F.22	0.039
7	F.07	0.255	19	F.15	0.008
8	F.04	0.241	20	F.23	0
9	F.05	0.227	21	F.16	0
10	F.03	0.208	22	F.12	0
11	F.06	0.206	23	F.11	0
12	F.17	0.201			

4.2.3 Classification Results

In this subsection, we describe the results of the experiments. We show the most influential features and the best classification algorithms.

4.2.3.1 Influence of Features

Table 4.5 shows InfoGain values for various features. 19 out of 23 proposed features are considered as influential predictors (InfoGain > 0). F.09 is the highest ranked feature splitting lines in the rectangle. Next, F.20 is an influential feature. Also, F.01 denotes rectangle coverage is one of the most vital features.

4.2.3.2 Classification Algorithms Performance

We evaluated the classification algorithms by measuring specificity and sensitivity over ten runs for the feature set. Table 4.6 shows the evaluation results. Table 4.6 shows the sensitivity and specificity scores. In term of sensitivity, (RF) is the best classifier with 96% of UML class diagram images correctly classified. On the other hand, based on specificity, (LR) performed the best with 91% of correctly classified non-UML class

Table 4.6: *Sensitivity and Specificity Scores for all Features*

	DT	J48	LR	RF	RT	SVM
Sensitivity	0.919	0.925	0.902	0.959	0.92	0.924
Specificity	0.895	0.901	0.914	0.904	0.901	0.89

diagram images. The standard deviations on the results are relatively small (0.01-0.05), which indicates the results are reliable.

The confusion matrix in Table 4.7 illustrates the classification result generated by applying the (LR) algorithm. From 1300 images, 1183 images were classified correctly. 596 out of 650 UML class diagram images. Also, 587 out of 650 non-UML class diagram images were correctly classified.

Table 4.7: *Confusion Matrix – (LR) classification*

Actual Results	Prediction Results	
	Y	N
Y	596	54
N	63	587

4.2.4 Image Processing Time

The average processing time is 5.84 second per image. Images that have bigger sizes and large amounts of lines need more time to be processed.

4.3 Extracting UML Models From Images

In this section, we explain our recognition technique for extracting UML models stored in image formats. We propose our recognition tool (Img2UML) [52][53] that can extract model information from three types of UML diagrams: UML class diagram, sequence diagram, and use case diagrams. Img2UML stores the extracted information into XML Metadata Interchange (XMI) format. XMI are XML files that store all information of a UML model. XMI files can be loaded into a CASE tool which they were created. However, each CASE tool uses its XMI structure as a result of which is not necessarily recognizable by another CASE tool. In the next subsection, we show the overview of the Img2UML tool.

4.3.1 Approach of Img2UML

For recognizing the three types of UML diagrams, we have to build a system that able to recognize shapes, symbols, lines, and text. Also, we need to identify the role of each diagram element in the diagram. Recognizing text in UML diagrams is essential to make the tool practical. Finally, we need to store all extracted information from UML images into XMI files, which is compatible with current UML CASE tools for reusing and editing models.

4.3.1.1 Input Images

Img2UML can read most images formats that are exported by most of UML CASE tools such as: jpg and png. These images can be black and white or colored images. Img2UML can load one image at a time, or a set of multiple images as input. The images in the set may be of varying size, type and color, and may originate from different UML tools. The Img2UML tool converts all images to the BMP format. The remaining image processing is subsequently standardized to work with this one image type. After conversion to BMP, a grayscale filter is applied. The next step in the process is segmentation.

4.3.1.2 Image Processing Algorithms

We use some algorithms supported by AForge.Net [54] with some modification, such as the algorithm for detecting rectangles. We create our algorithms for detecting different styles of lines, which is considered as the most difficult part of the recognition technique.

Segmentation is the main part of the image processing, which is used to analyze the representation of an image. Until now, there is no general solution for image segmentation [55]. We improve the quality of the processed images by applying suitable filters such as: Grayscale, Sharpen, GaussianSharpen, and Threshold. Then we use our segmentation algorithm and geometric-based approaches to enhance the accuracy of the recognition.

We create different algorithms for detecting horizontal, vertical and diagonal lines. Those lines can be solid or dashed lines. We notice that we detect each solid line type on a different copy of the original image, and we use that copies for detecting dash lines. For example, we use a copy for an image for detecting solid horizontal line. Then, we use the same of this copy to detect dashed horizontal line. Moreover, the same for other types of solid and dashed lines.

Detection of the left-leaning diagonal lines Figure 4.3 shows three left-leaning diagonal lines and how it represents in pixels. We created a general algorithm for detecting these three types of lines. After applying GauasianSharpen, Grayscale and Threshold filters on an image, we start reading the image from the top left point pixel (0,0). When we find a white pixel, we call this pixel the starting point and then:

1. We start looking for another white pixel starting from the new column and row values by incrementing each one with (+1).
2. After that, we search within a range of 10 column pixel to find another white pixel. If we find a white pixel (we call it the end point), then we go to step (1).
3. If we do not find a white pixel with the 10-pixel range, we back to the starting point, and add (+1) to the row pixel, and go to the step (2).

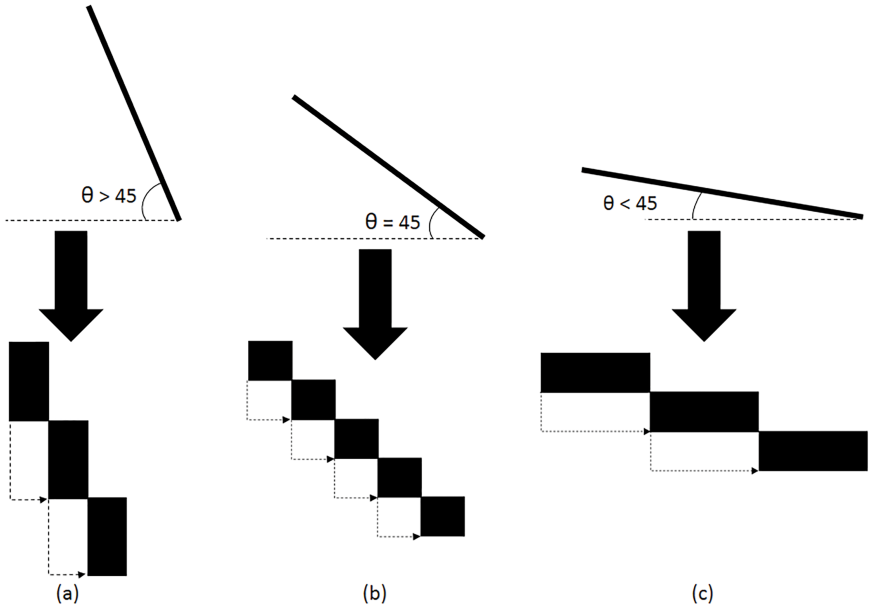

Figure 4.3: *Three different left-leaning diagonal lines and how they look like in pixels*

4. If we could not find a white pixel, we count the distance between the starting point and the end point. If the length of the line is more than 20 pixels, we store the line in the lines_array, which it is the array that contains all lines detected in the image. Then, we change the color of the detected line into black to skip detecting it again later, and go to the step five. Otherwise, if the length of the line is smaller than 20 pixels, we go to the step five.

5. Go to the starting point and increment the column pixel (+1), and start search for another white pixel (go to step (1) until we scan the whole image pixels).

Detection of the right-leaning diagonal lines For the detection of the right-leaning diagonal lines, we invert our algorithm to start the search for the black pixels from the top right point instead the top left point. We take care of the changes that should be done in the algorithm of detecting left-leaning diagonal lines to make it work for detecting right-leaning diagonal lines.

Detection of the horizontal lines Figure 4.4 shows the flowchart of the algorithm that used for detecting horizontal lines in the UML class diagrams. Algorithm 1 shows the pseudocode of the algorithm. We notice that after we detect a horizontal line, we remove it from the image by converting its color onto black. Removing solid horizontal lines help to avoid false-positive detection of horizontal dashed lines.

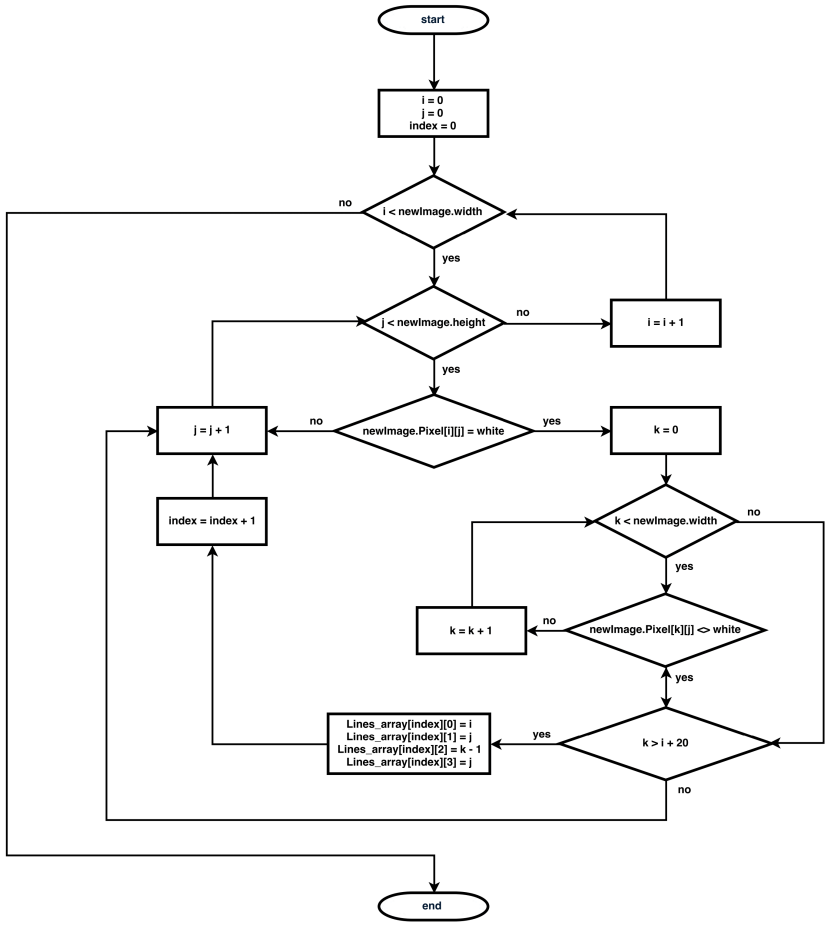

Figure 4.4: *Flowchart of the horizontal lines detection algorithm*

Detection of the dashed horizontal lines Figure 4.5 shows the flowchart of the algorithm that used for detecting horizontal lines in the UML class diagrams. Algorithm 2 shows the pseudocode of the algorithm.

Detection of the connected lines Figure 4.6 shows a class diagram, where some classes such as (Order) and (OrderGroup) are connected with one horizontal line. Other classes such as (Customer) and (Order) are connected with two horizontal lines and one vertical line. Therefore, because we detect each type of line separately, we need to detect the connected lines that are detected in an image.

We use lines_array to store all detected solid lines (horizontal, vertical, diagonal). Therefore, after detecting all possible solid lines in an image, we investigate the

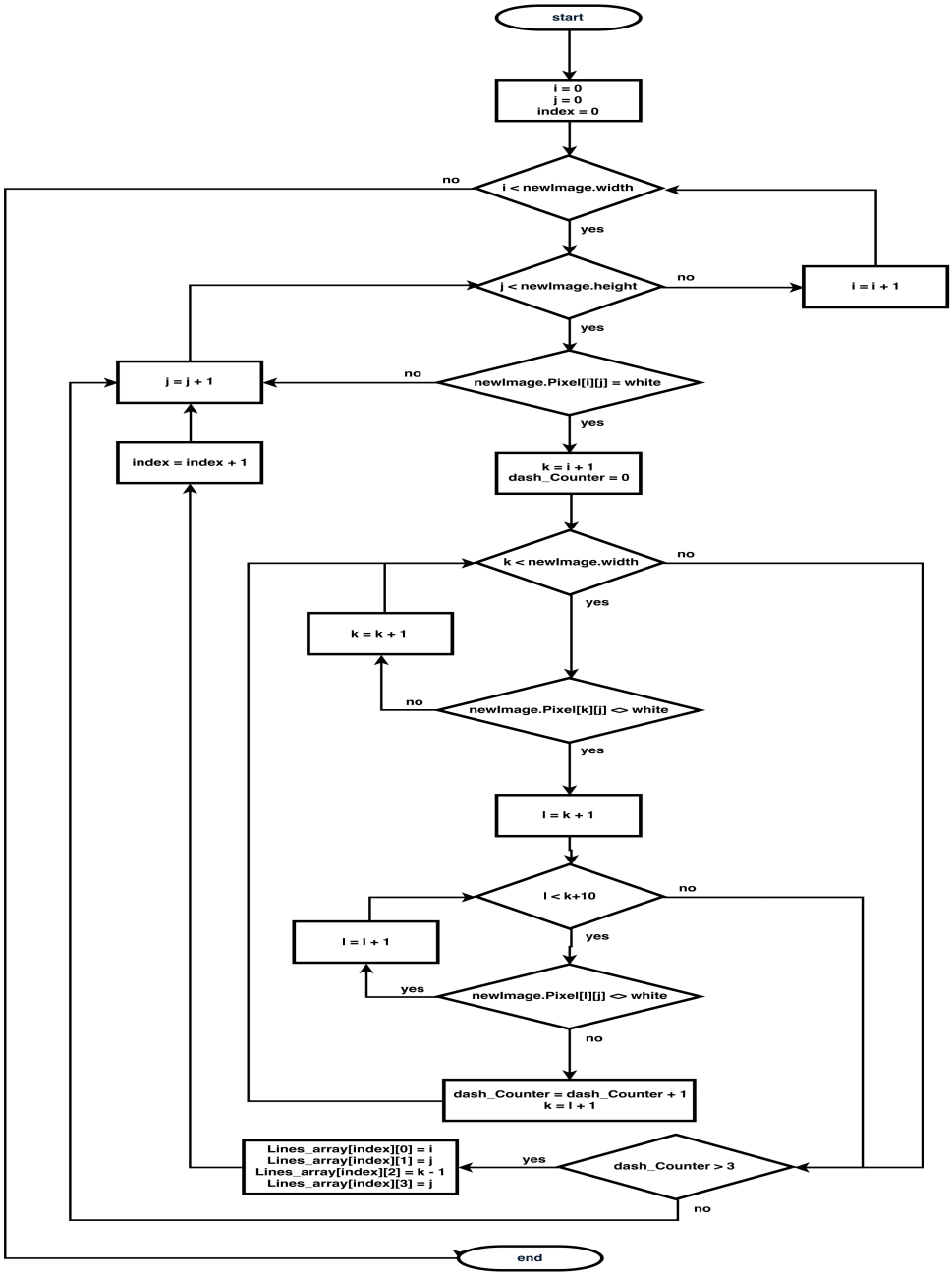

Figure 4.5: *Flowchart of the dashed horizontal lines detection algorithm*

Algorithm 1 Detect Horizontal Lines

```
1:  index ← 0
2:  for i = 0 → Image.width do
3:     for j = 0 → Image.height do
4:        if Image.Pixel(i,j).color = white then
5:           for k = i+1 → Image.width do
6:              if Image.Pixel(k,j).color ≠ white or k = Image.width-1 then
7:                 if k ≥ i + 20 then
8:                    Lines_array(index, 0) = i
9:                    Lines_array(index, 1) = j
10:                   Lines_array(index, 2) = k – 1
11:                   Lines_array(index, 3) = j
12:                   index = index + 1
13:                end if
14:             end if
15:          end for
16:       end if
17:    end for
18: end for
```

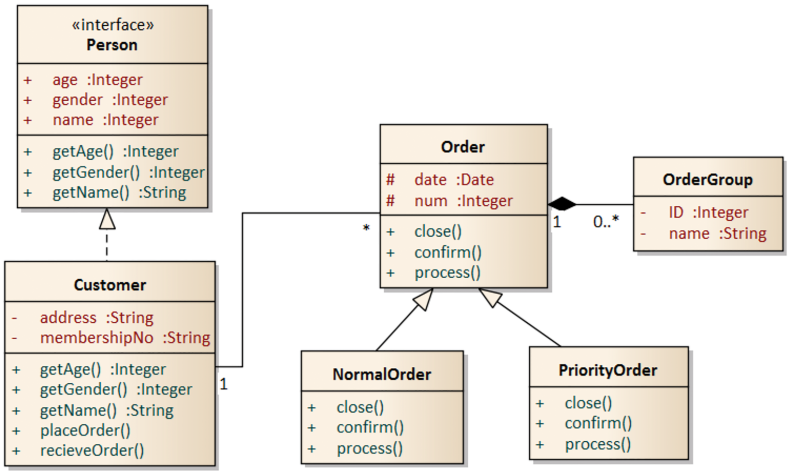

Figure 4.6: *UML Class Diagrams Example*

lines_array to find connected lines. We are matching the start point of each line with end points of the others in the line_array. When there is matching, we replace the end of the compared line with the end of the matched line, and we delete the matched line from the line_array. Then we start to compare the new line with other lines in the

Algorithm 2 Detect Dashed Horizontal Lines

```
1: index ← 0
2: for i = 0 → Image.width do
3:    for j = 0 → Image.height do
4:       if Image.Pixel(i,j).color = white then
5:          dash_Counter = 0
6:          for k = i+1 → Image.width do
7:             if Image.Pixel(k,j).color ≠ white or k = Image.width-1 then
8:                for l = k+1 → k+10 do
9:                   if Image(l,j).color = white then
10:                     Break
11:                  end if
12:               end for
13:               if l < k+10 then
14:                  dash_Counter = dash_Counter + 1 k = l+1
15:               else
16:                  if dash_Counter > 3 then
17:                     Lines_array(index, 0) = i
18:                     Lines_array(index, 1) = j
19:                     Lines_array(index, 2) = k – 1
20:                     Lines_array(index, 3) = j
21:                     index = index + 1
22:                     Break
23:                  end if
24:               end if
25:            end if
26:         end for
27:         if dash_Counter > 3 then
28:            Lines_array(index, 0) = i
29:            Lines_array(index, 1) = j
30:            Lines_array(index, 2) = k – 1
31:            Lines_array(index, 3) = j
32:            index = index + 1
33:            Break
34:         end if
35:      end if
36:   end for
37: end for
```

line_array.

We notice that the detection of each type of lines separately (horizontal, vertical, left-leaning diagonal and right-leaning diagonal), then connected them together works better regarding accuracy and time rather than other algorithms. One of the most reason of this is when an image (diagram) has crossing lines.

4.3.1.3 Recognition of UML models stored in image formats

In this section, we are describe the image processing techniques used for each type of UML diagrams. We use the Aforge.NET framework image processing library [54].

UML Class Diagram Figures 4.7 and 4.8 show a class diagram before and after the recognition using Img2UML, respectively. After applying Grayscale and Threshold filters, images follow four consecutive processing:

1. *Detecting classes in the images*: We detect rectangles in the images. Rectangles are detected by using Aforge.NET Framework image processing library after some quality improvements the recognition algorithm. For example, we change the gap threshold between the connected lines that create a rectangle. This threshold changes automatically based on image resolution and image, size and length of the expected line.

2. *Recognizing text in the classes*: Rectangles that represent classes may have three different rectangles (parts), which represent areas for class name, attributes, and operations respectively. So we try to detect two or three rectangles contained in large rectangles. Finally, we use an OCR library for recognizing text inside the detected rectangles. We explored two OCR libraries, Microsoft Office Document Imaging (MODI) and tesseract-ocr. The results showed that MODI gives more accurate results than tesseract-ocr. Therefore, we used MODI.

3. *Detecting relationships*: Dependencies between classes are depicted as lines which connect rectangles that represent classes. Across different images, we find many different styles of drawing such connecting lines: straight, hooked (horizontal and vertical), diagonal, curved, solid and dotted. Our tool cannot detect curved lines. Detecting lines is the most difficult parts the recognition.

4. *Detecting UML class diagrams symbols*: in this part, we detect the types of the detected relationships. There are four types of relationships: associations, generalization, dependency, and realization. The association relationship has four kinds associations: Association, direct association, aggregation, and composition. We need to detect six symbols for determining seven types of relationships. These symbols are small geometric shapes.

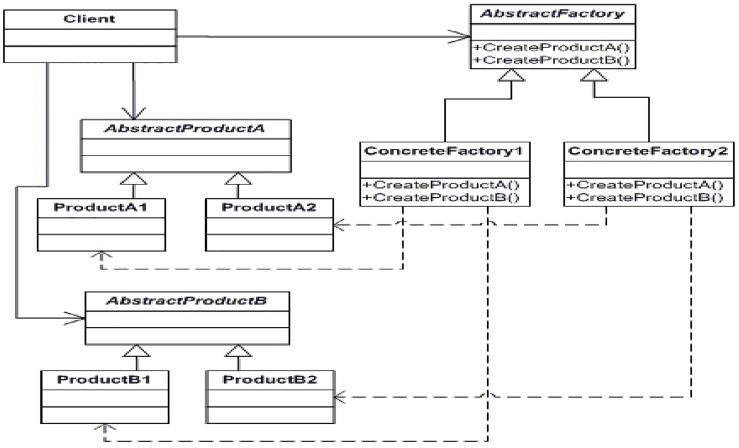

Figure 4.7: *UML Class Diagrams before the recognition*

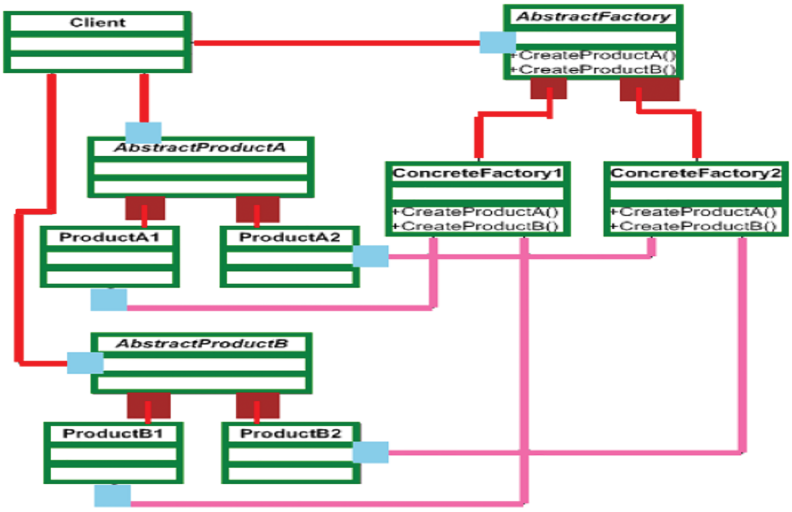

Figure 4.8: *UML Class Diagrams after the recognition*

UML Sequence Diagram Figures 4.9 and 4.10 show a sequence diagram before and after the recognition using Img2UML, respectively. The recognition processing is consists of four consecutive parts:

1. *Recognizing lifeline's headers*: lifelines headers are represented in rectangles, so we detect the rectangles in the images.

2. *Recognizing the actors*: it is popular that some practitioners use a stick man for actors instead of rectangles. To recognize the stick man, we search for a circle and

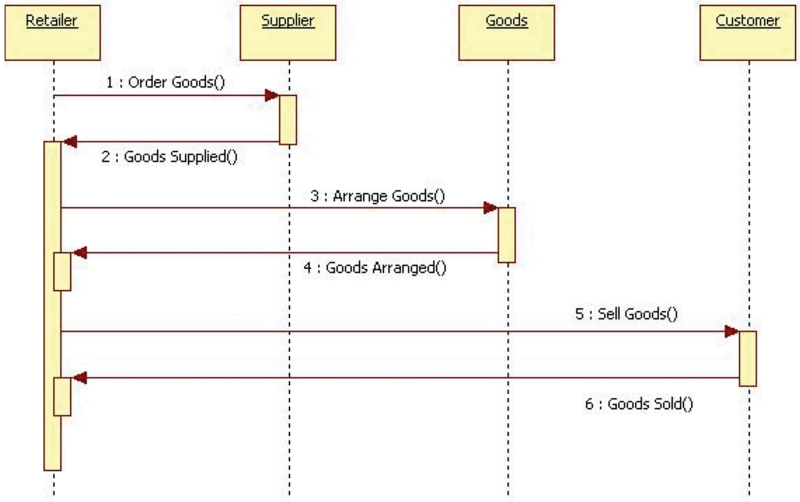

Figure 4.9: *UML Sequence Diagram before the recognition*

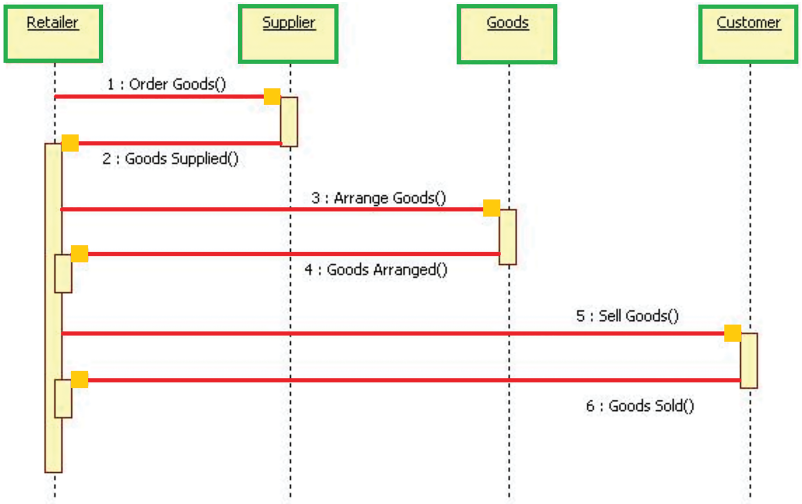

Figure 4.10: *UML Sequence Diagram after the recognition*

then check if the circle is at the top of a vertical line.

3. *Message's lines*: Lines with an arrow at one end of the line pointing to a destination lifeline symbolize messages. The UML specification does not recommend that lines should be horizontal. However, we have realized that in practice these lines

are often horizontal. Therefore, in our tool we only focus on the horizontal lines. We recognize horizontal solid and dashed lines.

4. *Recognizing arrows*: arrows of the message's lines on a sequence diagram provide information about the direction of the message, which determines the source and the target lifelines of a particular message. Furthermore, the shape of an arrow i.e. solid or open arrow is a determination of the type of the message.

5. *Message's type*: in the sequence diagrams, there are many types of messages: synchronous call, asynchronous call, return, create, and destroy messages. We recognize messages types using results of the recognition of lines and corresponding arrows and message's name. For example: if a line is solid and its arrow is solid, the message is considered as a synchronous call. Another example: if the name of the message is "create" the type of the message is considered as a create message.

6. *Recognizing text*: we examined MODI and tesseract-ocr. After testing these two technologies with several images, we found that MODI is more accurate.

UML Use Case Figures 4.11 and 4.12 show a use case before and after the recognition using Img2UML, respectively. The processing consists of four consecutive parts:

1. *Recognizing the actors*: actors are usually drawn as a stick man, or alternatively as a class rectangle. To recognize the stick man, we search for a circle and then check if the circle is at the top of a vertical line. If there is no any circle, we try to search for rectangles to denote the actors.

2. *Recognizing use cases*: use cases are represented as ellipses, so we detect ellipses in the images.

3. *Recognizing connectors*: the notation for using a use case is a connecting line between an actor and the use case. So we detect lines in the images. These lines can be straight lines or diagonal lines.

4. *Recognizing text*: texts can be the use cases names or actors' names. As usual, the use cases names are positioned inside their ellipses. Actors' names are represented by a text under the stick men or inside rectangles. We use MODI for detecting texts inside detected ellipses, and under the detected stick men or inside detected rectangles.

4.3.1.4 Generating XMI

After recognizing model information from images, such as classes, relationships, actors, ellipses, etc., we need to organize it in suitable data structure and file format. XMI is the

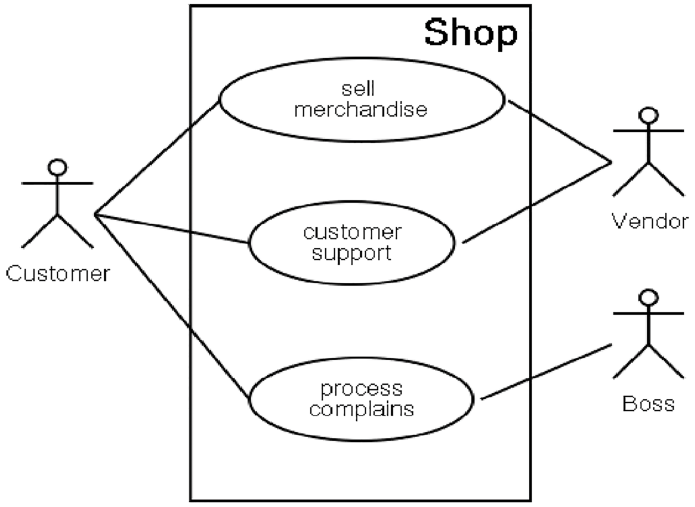

Figure 4.11: *UML Use case before the recognition*

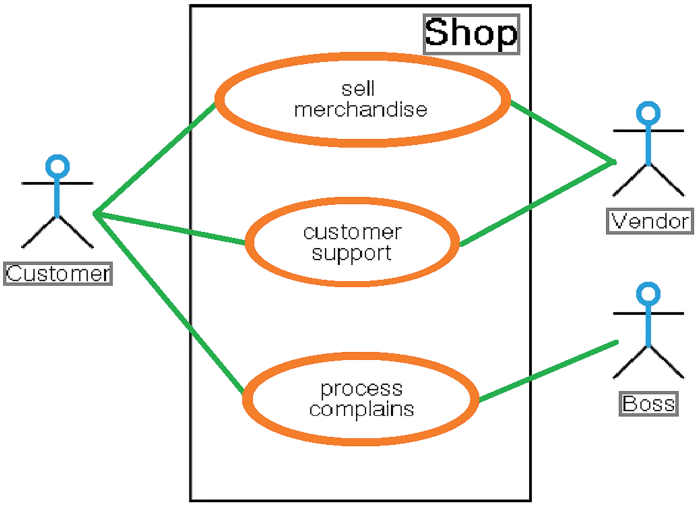

Figure 4.12: *UML Use case after the recognition*

most used format for this purpose. We choose XMI 1.1 for UML 1.3 Rose Extended for generating XMI files. Our generated XMI files are compatible with many current UML CASE tools such as Enterprise Architecture [56], Visual Paradigm [57] and StarUML [58].

4.3.2 Why UMLCrawler, UMLImgClassifier and Img2UML

We create separate tools with different approaches because we need them to work independently in a pipelined way. This pipeline saves more time, where the output of UMLCrawler is the input of UMLImgClassifier, and the output of UMLImgClassifier is the input of the Img2UML. UMLCrawler is faster than other tools then UMLImgClassifier in terms of the output images. UMLImgClassifier can classify more images than what Img2UML can do in a specific time because it just extracts specific features from images instead of extracting the whole information as is done by Img2UML. Therefore, the pipeline chain becomes: UMLCrawler is collecting UML images from the internet and saves it to the local drive. Then UMLImgClassifier classifies these images and removes the Non-UML class diagrams images. Finally, Img2UML extracts model information from images that are available on the local drive and generates XMI files.

4.3.3 Validation of Img2UML

For validating Img2UML tool, we performed three experiments on UML class diagrams, sequence diagrams and use cases. The validation process is done manually, by comparing class diagrams in images with class diagrams in XMI files visualized by StarUML CASE tool. It takes some time and effort. Therefore, we focus most of our attention on the validation of class diagrams.

4.3.3.1 UML Class Diagrams

For validating Img2UML for UML class diagrams, 500 class diagrams in different image formats are collected from the internet. These images vary in color, type, size and resolution.

As a result, the accuracy of the Img2UML system is: for classes, 95% of the rectangles that denote classes are recognized, for relationships it is 80% and for text recognition it is 92%. Many factors affect the accuracy of detection. The most important factor is image resolution. Problematic cases in the remaining 5% for class detection were related to image resolution, and some rectangles are crossed by some lines that can be considered as a bad layout. There are two main problems in detecting relationships. First: symbols that determine types of relationships. The main problem of detecting these symbols is their small size. Second: dashed lines, especially dashed lines.

4.3.3.2 UML Sequence Diagram

We validate Img2UML on 20 images randomly taken from our collection. Img2UML provides an average accuracy of 75% in the recognition of all elements in a sequence diagram. The accuracy of the recognition of lifelines is 74% and messages 76%. We stated from this experiments that extraction of elements name (using MODI) and

recognition of arrows for the identification of the type of messages are the most difficult information. The main reasons for these difficulties are low image resolution that makes the recognition of text and diagonal lines more difficult. From 100 of sequence diagram images we collected via our crawler, we found that:

- 97% of the sequence diagrams contain messages in the form of a horizontal line.
- 97% of the sequence diagrams contains lifelines in the form of a rectangle and 31% stick man. 8% contain lifelines in another form.

We replaced four from the twenty images selected for the validation. The new images do not contain diagonal lines lifelines are represented as rectangle or stick man. The results show that Img2UML provides an average accuracy of 85% in the recognition of the elements of sequence diagrams.

4.3.4 UML Use case

The test set contained 36 images we gathered randomly from our collection from the internet. The results of the object recognition are:

- Use cases: 91
- Actors: 89%.
- Relations Actor-use cases: 69
- Relation use cases-use cases: 85
- System border: 92%.

The objects and the characters are recognized very well except for the actor names because the location of the actor names is guessed. The main reason for the difficulties in recognizing relations between actors and use cases is low image resolutions. Most of these relations are diagonal lines, and sometimes they are not connected or not close enough to other objects (actor and use cases).

4.4 Related Work

There are many approaches for collecting models from the internet. For example, the search engine proposed by Lucrédio et. al. called Moogle [59]. The system consists of three parts: the model extractor, the searcher, and the user interface. The searcher is based on an open source search engine: Apache SOLR, which establishes an index for the model descriptor that contains all the necessary information of them. Moogle is a search engine for model files that are presented with text. What we want to achieve are for UML models that are presented as diagrams and stored in image formats, XMI

files or native UML CASE tools files. As there are already different search engines that deal with text, it is more difficult to convert images into text models and do the query. Another drawback is the searching method. Moogle uses indexes for model descriptors to implement the query process. It is a search within files. The efficiency will not be as good as querying a relational database.

Image classification denotes to the labeling of images into different categories. Lu et. al. [60] proposed major steps for image classification process. We follow the steps proposed by Lu et. al. for building our classifier.

Many researches are proposed for classifying images, for example classifying remote-sensing images [61]. Chart image classification also is a one of the most concerned topic [62]. To the best of our knowledge, there is no study about classifying UML diagrams images.

We can categorize engineering diagrams on how they were made into two categories: hand-made and computer-made via engineering tools (using predefined geometric shapes). Tools for hand-drawn images are called sketch tools.

Fu et. al. [48] illustrated two motivations of engineering diagram recognition. Firstly, some engineering diagrams in images in diverse design and education related scenarios are non-trivial. Secondly, engineering diagrams recognition enhances the supportive value of diagrams in design. They proposed methods for recognizing computer-made and hand-made engineering diagrams.

Yu et. al. [63] presented a system for recognizing a large class of computer-made engineering drawings such as flowcharts and electrical circuits. Their system does not support UML models as most systems for engineering diagrams recognition.

Diagram feature extraction is another important topic, and it is important for classifying images. Messmer et. al. [64] proposed a system for recognizing of sketched graphic symbols in engineering drawing. They combined pattern recognition techniques with machine learning concepts for learning and recognizing symbols in engineering diagrams.

Ablameyko et. al. [65] showed that the interpretation of engineering drawing is a complex and theory-weak process. They mentioned that systems supported this technology is still difficult to put into engineering applications.

Many methods are proposed for recognizing hand-drawn UML diagrams [66, 67, 68, 69, 70]. Most of these methods support UML class diagrams, and a few of them supports sequence and use case diagrams. We show that various researchers proposed different approaches for recognizing engineering diagrams in images and hand-drawn diagrams. Most studies related to UML diagrams are based on sketched diagram. The techniques used for recognizing sketching UML models cannot be carried over to recognize UML models in images. Thus because algorithms in sketching tools are based on information regarding the movement of drawing, which is not available in images.

4.5 Conclusion and Future Work

In this chapter, we present our developed tools to collect UML diagrams. The *UML-Crawler* can download a huge number of UML diagram stored in image format from the internet via Google Images. The *UMLImgClassifier* can classify UML class diagrams images from other images. Img2UML tool can extract model information from UML diagrams stored in image formats and generate XMI files. Img2UML eliminate the gap between pixel-based diagrams and engineering models. Any mistakes in the generated models (XMI) can be resolved by editing the models in a UML CASE tool because the generated XMI files are compatible with most current UML CASE tools. Engineers, developers, researchers, teachers and students may find these tools useful for collecting UML models, classifying class diagrams and extracting model information from UML Class diagrams stored in image formats. The validation shows that our classifier and Img2UML provide high accuracy for classifying UML class diagrams and convert UML class diagrams and sequence diagrams stored in image formats into UML models.

For future work, we plan to extend our classifier to support UML sequence diagram and use case diagrams. In addition, we are going to extend Img2UML to support other UML diagrams.

Chapter 5

Models-db.com: An Online Repository for UML Models

In this chapter, we present a repository for UML models. The repository contains models in various image formats and XMI format. This repository is the first corpus of UML models of this kind. The repository also contains more than 800 UML class diagrams, and also various types of meta-data. This repository aims to provide a good place for researchers and students to study and analyze UML models. The repository also supports the definition and execution of experiments that employ the UML models from the repository. In this chapter, we argue for the usefulness of this repository. We explain how the data was collected and the services that are offered by the online repository.

This chapter is based on the following publications:

- Bilal Karasneh, Michel R. V. Chaudron. **Online Img2UML Repository: An Online Repository for UML Models** *In Proceedings of the 3rd International Workshop on Experiences and Empirical Studies in Software Modeling (EESS-MOD@MODELS 2013), pages 61-66, Miami, USA. 2013.*
- Bilal Karasneh, Michel R. V. Chaudron. **Img2UML: A System for Extracting UML Models from Images** *In Proceedings of the 39th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2013), pages 134-137, Santander, Spain. 2013.*
- Truong Ho-Quang, Michel R. V. Chaudron, Ingimar Samúelsson, Jóel Hjaltason, Bilal Karasneh, Hafeez Osman. **Automatic Classification of UML Class Diagrams from Images** *In Proceedings of the 21st Asia-Pacific Software Engineering Conference(APSEC2014), pages 399-406, Jeju, Korea. 2014.*

In the previous chapter, we introduced the Img2UML tool, which can extract model information from three types of UML models stored in image formats, and store them in XMI file format. A large collection needs a way to search for models based on their contents. For example, to search for class diagrams that contain the class name "reservation". We consider searching in XMI files as an inappropriate way because it is based on file systems, which is most related to an unstructured data store for sorting arbitrary, probably unrelated data. Another reason is that searching in databases is more efficient than searching in file systems. Therefore, we present a novel repository for UML models. The repository is available online at:

http://models-db.com/

The repository includes images of the diagrams, XMI files (the tool independent representation of UML models) and design metrics for each available UML diagram. In the current repository, we only publish UML class diagrams because our focus is on UML class diagram.

5.1 Related Work

It is essential to have a large collection of sample data in many research fields. Corpus studies are widely common and used because they play an important and crucial role in the scientific process. For example, in linguistic, corpus linguistics is the study of language based on samples (corpora). Corpus linguistic has developed to support empirical investigations of language variation and use. There are many examples of corpora such as British National Corpus (BNC)[1] and the International Corpus of English (ICE)[2]. There are special tools used to analyze a corpus and search for certain words or phrases. The Wordsmith Tool [71] [71] is software in widespread use that allows performing lexical searches and various types of statistical analysis. In Biology, where there are many repositories available. There are repositories for biospecimens e.g. biobanks and tissue banks. Specimen Central[3] is a comprehensive directory of biospecimens repositories. Another kind of bio-repositories such as Cell Image Library [4] and CellML[5]. Cell Image Library contains a variety of organisms in images and video formats and CellML contains biological models.

In Software Engineering, there are many repositories published for different purposes. Repositories are important because of the need of the community of software engineering for advanced systems supporting the reuse of software artifacts, and adoption of software management tools as a service. Rodriguez et. al. [38] classify

[1] http://www.natcorp.ox.ac.uk/
[2] http://ice-corpora.net/ice/
[3] http://specimencentral.com/
[4] http://www.cellimagelibrary.org/
[5] https://www.cellml.org/

repositories in software engineering and discuss their general problems. Rocco et. al. [72] show an overview of model repositories that contains different modeling languages such as SysML and UML. They present some technical and non-technical challenges of models repositories. As an example of a technical challenge, model repositories do not support advanced query mechanisms, which are crucial for retrieving models. For instance, searching for models based on domain type or development phase. A non-technical challenge concerns the licensing related to the shared artifacts. Basciani et.al. [73] show that in different application domains like biology and source code development, repositories are already a reality, and they are continuously used to share, learn, reuse, and improve artifacts. They show that model repositories in software engineering are still far from a concrete adoption of repositories in other domain.

Nowadays, few UML repositories are available. Some repositories are supported by UML CASE tool vendors [56][57] on a commercial basis. These repositories only support their own file - native CASE tools format. In addition, there is no open access for models created by others tools. Because of these obligations and associated costs, this kind of repositories is not considered attractive from the perspective of academic research.

Another kind of repositories is the general model repositories. France et. al. [36] proposed a repository for model- driven development called ReMoDD, which contains many documented case studies. This repository is a great asset for researchers where they can find many examples of models as well as research studies. However, UML diagrams in the ReMoDD are stored as files. Therefore, these models are not searchable. In order to view a model, one has to download it and then open it using a compatible CASE tool. In addition, some of the UML models are stored in PDF files. However, not all PDF files in the repository contain UML models. Therefore, users need to open these PDF files and search manually for UML diagrams.

5.2 Usefulness of the Repository

Our interest is in software design. Our repository contains more than 800 class diagrams related to different application domains. This set of models offers opportunities for researchers.

The repository can be used to analyze class diagrams, measure qualities, study common flaws and their frequency of occurrence, compare quality-models, etc. The availability of different versions of models for one software system provides an opportunity to study the evolution of versions of class designs. The repository can also be used for studying UML class diagrams in software engineering courses. Students can reuse available class diagrams, share their knowledge, and engage in discussions about models.

The repository contains design metrics for all available class diagrams. Metrics are important to understand, steer, and control the development of complex software

[74]. Moreover, metrics can be used as indicators of software quality. The availability of design metrics for a large collection of UML class diagrams is a great source for studying the quality of UML designs. It also enables the creation of benchmarks (e.g. related to design problems).

The repository is searchable; it offers functionality for querying and searching for models based on different keys such as model information (class name, attributes, etc.). Models in the repository can be classified and analyzed automatically by using queries. For example, class diagrams which contain a high number of classes, a high number of relationships (dependency or inheritance), some design pattern, or some design anti-patterns. These queries can show common characteristics of class diagrams.

Our repository facilitates answers to many research questions:

- What are common patterns in practice?
- What are common anti-patterns in practice?
- Can we find quantitative characteristics of UML models (e.g. relation between size and max. coupling)?

We present some of these characteristics of class diagrams in chapter 6.

5.3 Data Collection Approach

In this section, we discuss the difficulties of collecting UML models and our approach to perform the collecting process.

5.3.1 Difficulties

Currently, there are no open repositories for UML models because the lack of the availability of UML models from both commercial and open source projects. Both suffer from the absence of a common representation and file organization of UML models. The confidentiality of commercial software development is the main reason. Companies are reluctant to share their models. Therefore, collecting UML models become difficult, and the empirical research of UML is challenging.

There is a large variety of representations of UML models in both graphical format and in terms of XMI formats by different UML CASE tools. Because of the large variety of representations of UML models, collecting UML models is challenging. Furthermore, there is no open technology for creating UML-repositories as there exist for source code (such as version management tools like github.com and sourceforge.net).

In our proposed repository, we start focusing on one type of UML diagrams, which is UML class diagrams. Class diagrams are ubiquitous in UML modeling; they are the most important structural model of the UML as they show the static description of the system regarding classes, relationships and constraints in the relationships [75]. Class

diagrams are very important when engineers need to understand the basic structure of a system, e.g. when a new engineer, that is unfamiliar with a system, needs to maintain it. Class diagrams are prevalent within industry and academia, where model-driven development is becoming a common practice. In addition, it is widely agreed that class diagrams have become an integral part [76][77].

The selection of class diagrams is based on the importance of the diagrams in software development and its availability. Next, we discuss our collecting approach.

5.3.2 Collecting Approach

Our aim is to collect UML class diagrams and their design metrics. Diagrams have been collected in different ways, and we collect design metrics for each class diagram. Next, we illustrate how we collect models, design metrics and challenges of the collecting approach.

5.3.2.1 Collecting Models

We collect models in a variety of ways:

A. *UMLCrawler*: UML models stored in images can be found in fair numbers of software documents and on the internet. We use *UMLCrawler* for collecting UML class diagrams from the internet. We also look for UML class diagrams in source code repositories, to have both the design and the source code. We use Img2UML system tool to convert UML class diagrams in images to XMI.

B. *Collaboration*: We ask people from different universities to share models that they have in any file format (CASE tool format, images or XMI). Most of the models we have received are created by different CASE tools, such as Enterprise Architecture and StarUML.

C. *Literature*: We are collecting datasets of UML class diagrams from the scientific literature – e.g. when used in experiments. We find most of class diagrams are published as images. Most of these models are used in controlled experiments. We use Img2UML to convert models store in images into XMI files.

D. *Users*: We ask the repository users to upload models that they have. We have a special page on the repository website for uploading models in images, XMI, and archive files (Zip files)[6].

5.3.2.2 Collecting Design Metrics

We use SDMetrics [78] to compute design metrics for all models in the repository. The SDMetrics tool uses an XMI file as an input and exports design metrics as CSV file.

[6] http://models-db.com/Upload.aspx

For each model, we compute 23 metrics that are provided by default by SDMetrics. In addition, we introduce six new metrics, which are:

1. maximum number of attributes.
2. maximum number of operations.
3. maximum number of coupling.
4. maximum number of children.
5. maximum number of depth of inheritance.
6. maximum number of class to leaf depth.

From the repository, we can investigate various relationships between these metrics through querying. In chapter six we show some sample investigations.

5.3.2.3 Challenges of the Collecting Approach

Since most UML class diagrams are collected in image formats, *UMLCrawler* and Img2UML system tool are the ambassadors of the collecting approach. Therefore, the challenge of the collecting approach is related to the limitation of both tools. The classifier [79] reduces the limitation of *UMLCrawler*, with 96% correctness for classifying UML class diagrams and 91% for non-UML class diagrams. For Img2UML system tool, each UML class diagram in image format converted to XMI, a manual check was performed to verify that all models information was included correctly in the XMI. Faults in the image recognition are corrected manually. We find that the success of the image recognition depends on the resolution of the image. For class diagrams, recognition of classes (rectangles) works very well, but the problem is in text recognition. The success of recognizing text (i.e. names of classes, operations) decreases with the resolution of images.

5.4 Repository Overview

The repository is continually increasing in size. We take a snapshot of the repository on the 24Th November 2015. We can classify models in the repository into four categories: experiments, projects, students and web models. We collected models from 55 experiments, 67 projects, students and from the internet. Table 5.1 shows a summary of the models in the repository.

Figures 5.1, 5.2, 5.3 and 5.4 show the distribution of models size based on the number of classes for experiments, projects, students and web models diagrams respectively. From these numbers, we observe that the diagrams size from experiments, projects and web models cover comparable size-ranges. Only models from students seem to differ

Table 5.1: *Summary of Models in the Repository*

Category	Number of Models	Range of models Size
Experiments	181	[2,50]
Projects	134	[1,50]
Students	87	[6,18]
Web Models	560	[1, 43]

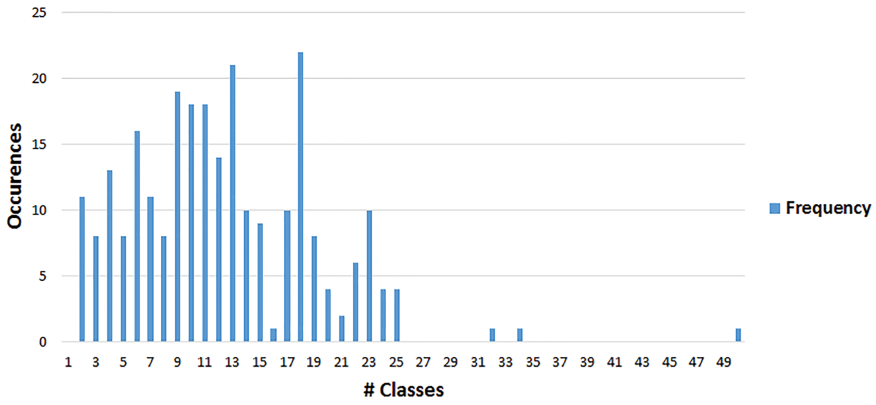

Figure 5.1: *Distribution of experiment diagrams size*

somewhat: they are not very small (at least 6 classes) but also not very large (at most 18 classes). The graphs of these size distributions also show that none of these categories contains any significant number of diagrams that contain more than 30 classes.

5.5 Repository Schema

The repository database contains many tables, 12 of them are related to models (the information extracted from XMI file). Figure 5.5 shows the structure of these 12 tables that are related to class diagrams. The database contains many other tables that are not shown in the Figure 5.5 We give a brief explanation of each table in the repository:

- **image_Table**: This table contains image path and information about the image such as height and width.
- **xmi_table**: This table contains XMI files path and related image IDs.
- **class_Table**: This table contains class names and the related XMI file.
- **attribute_Table**: This table contains attribute names and the related class IDs.

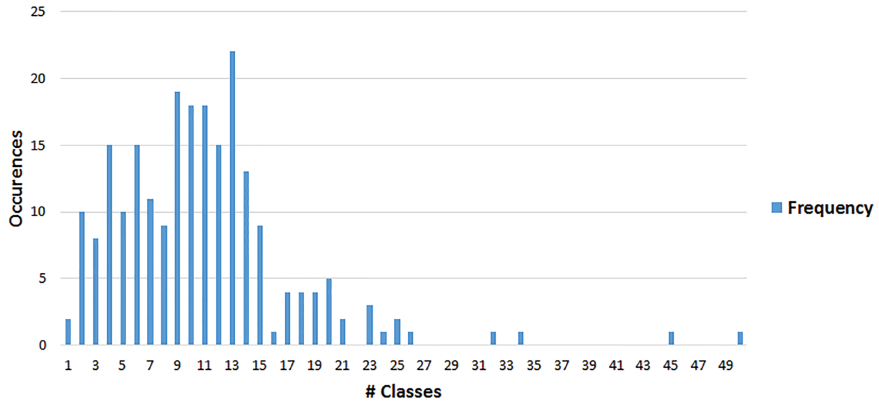

Figure 5.2: *Distribution of project diagrams size*

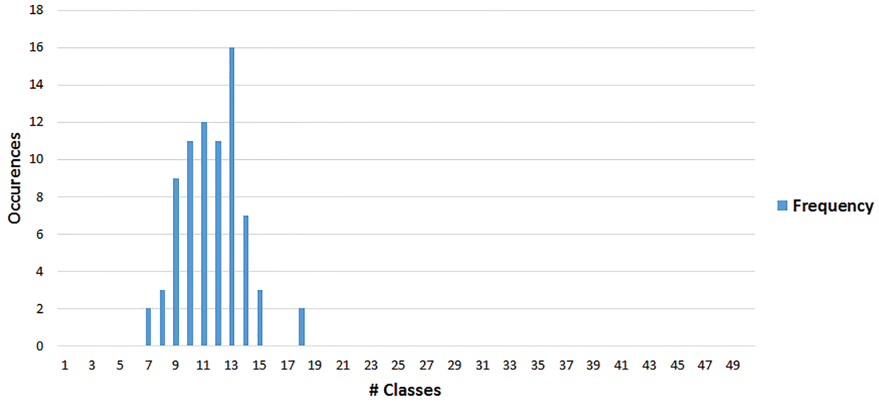

Figure 5.3: *Distribution of student diagrams size*

- **operation_Table**: This table contains operation names and the related class IDs.
- **association_Table**: This table contains association IDs and the related XMI IDs.
- **associationEnd_Table**: This table contains IDs of the classes connected via association relationship, and the related XMI IDs.
- **dependency_Table**: This table contains IDs of the classes connected via dependency relationship, and the related XMI IDs.
- **generalization_Table**: This table contains IDs of the classes connected via generalization relationship, and the related XMI IDs.

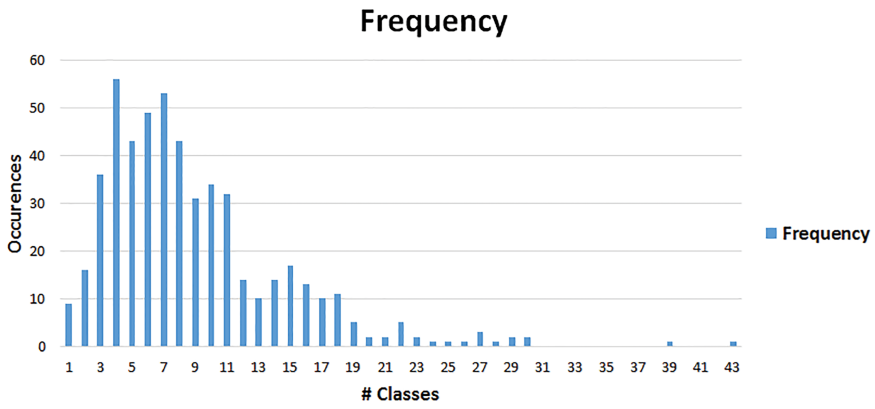

Figure 5.4: *Distribution of web diagrams size*

- **realization_Table**: This table contains IDs of the classes connected via realization relationship, and the related XMI IDs.
- **xmi_Data_Table**: This table contains XMI ID and design metrics related to the diagrams.
- **answers_Table**: This table contains question IDs, user IDs of those who defined the questions, all user answers and their IDs.
- **questions_Table**: This table contains questions, user IDs of those who defined the questions and questions URL.
- **uploaded_Images**: This table contains users' uploaded images and user IDs of those who uploaded the image. If a user is not registered, "anonymous" is added instead of user ID.
- **uploaded_XMI**: The table contains users' uploaded XMI files and user IDs who uploaded the image. If the users are not registered, "anonymous" is added instead of user ID.
- **uploaded_Files**: The table contains user uploaded files such as Zip files that contain many UML diagrams in image formats, many XMI files and/or both images and XMI files. Users can upload software documents as a project, when they want to publish and share with others. Moreover, the table contains user IDs who uploaded the image.
- **evaluation_Table**: This table contains user evaluations for six quality attributes related to the models available in the repository. These quality attributes are understandability, layout, extensibility, modifiability, completeness and correctness.

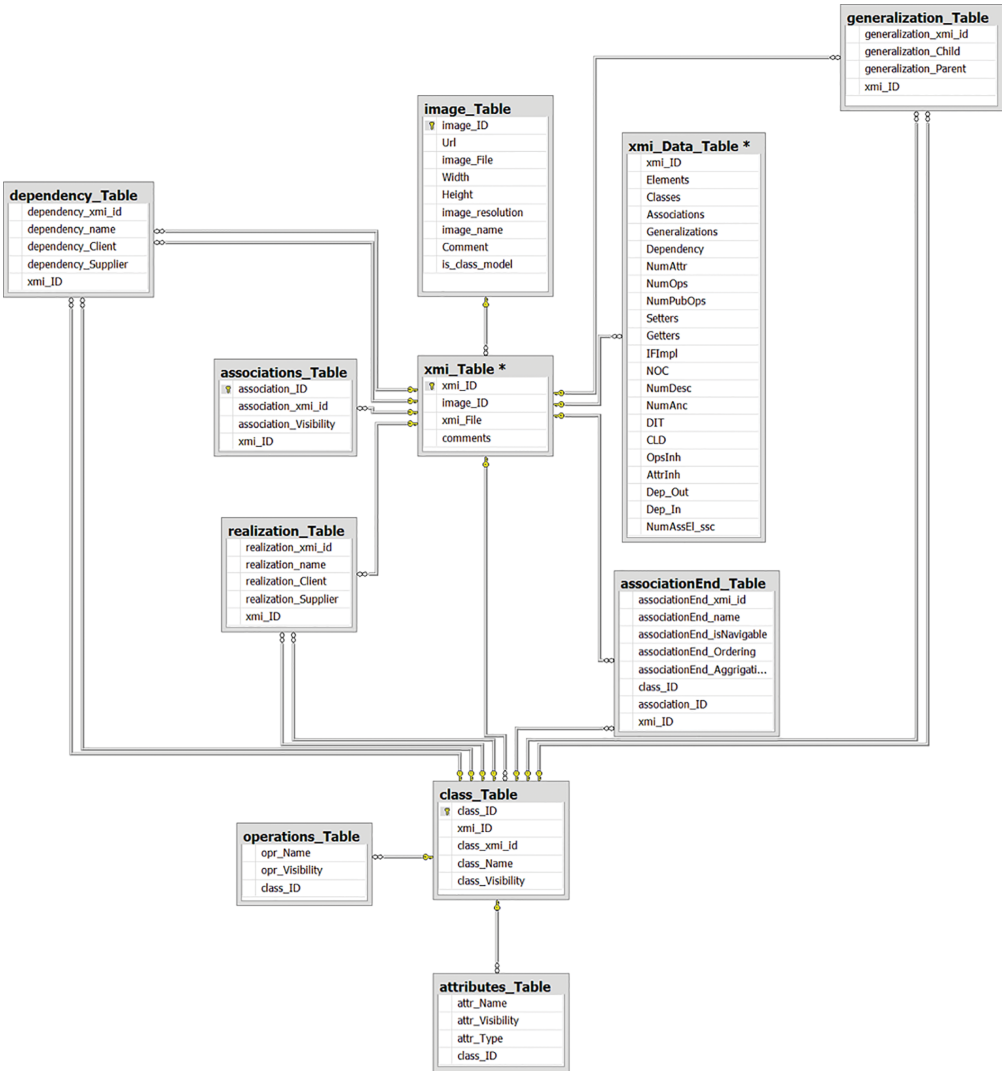

Figure 5.5: *Database Schema*

The table also contains IDs of the users who did the evaluations, image IDs for the models and user comments about each quality attribute. Users can use the comments field to explain their evaluation.

- **queries_Table**: This table contains all queries that users made for searching models. The table also contains user IDs.

- **user_Table**: The table contains user IDs and user names.

- **visited_Model_Table**: The table contains all models visited by users. The table contains image IDs and user IDs and the time of visiting.
- **visitor_Table**: This table records the activity of all users such as questions, answers, evaluations and their models.

5.6 Repository Services

The repository not only provides a collection of UML diagrams and supports searching for models. In addition, the repository provides services for creating experiments and online exams. Moreover, users can ask questions about any models and share their questions with others who can answer these questions or give them some feedback. The repository also offers user visualization services for the results of their queries.

5.6.1 Searching for Models

UML diagrams in the repository are searchable, where users can search for them using:

1. Class names.
2. Attribute names.
3. Operation names.
4. Design metrics.

The operators (+) and (&) and the keyword (and) can be used to search more than one class, attribute or operation. For example: "reservation and patient". In this example, we search for models that contain two classes (reservation) and (patient). This way can make the search more specific, because in the same previous example searching keyword for class (reservation) can be found in hospital and hotel class diagram. Using design metrics is another way of searching, where users can perform their search based on:

1. Number of classes.
2. Number of attributes.
3. Number of operations.
4. Number of elements, where the elements are the sum of the number of classes, attributes, operations and relationships in the diagram.
5. Number of (NOC), which are the number of children in the diagram.
6. Number of (DIT), which is the depth of inheritance in the diagram.

7. Number of maximum coupling in the diagram.

For the design metrics properties, users can search using five choices:

1. Any, which means any number.
2. Greater than, where users have to specify a number, and the results will be greater than or equal to the specified number.
3. Less than, where users have to specify a number, and the results will be less than or equal to the specified number.
4. Between, where users have to specify two numbers, and the results are between or equal these two specified numbers.
5. Exactly equal to a specified number (entered by user). The user can use one or more properties for searching, and the user can filter the results in the result page.

From these features of querying models, users can easily select models, and access the descriptive details of the models. These features are important and useful. However, we note that such features are not available in code-oriented repositories.

Next, we are going to discuss the results of user queries.

5.6.1.1 Diagrams Search Results

The result web page contains the result of user queries. The page contains diagram(s) retrieved, associated images, design metrics and diagram categories (experiment, project, students, and web model). From the result page, users can:

1. Filter design metrics they want to show from all available design metrics.
2. Select some or all class diagrams to ask question(s) about them. Figure 5.6 shows the results page after searching for the class name "reservation".
3. Select one or more class diagrams to find the similarity between them based on design metrics. We explain this feature later in section 5.6.3.
4. Charting the diagrams in the result page based on their design metrics. We also explain this feature later in section 5.6.4.
5. Show the diagram reference (URL), from where it is collected.
6. Navigate to the page where users can evaluate quality attributes of the selected diagram.
7. Download XMI files.
8. Export the results into MSWord, MSExcel, CSV or as PDF files.

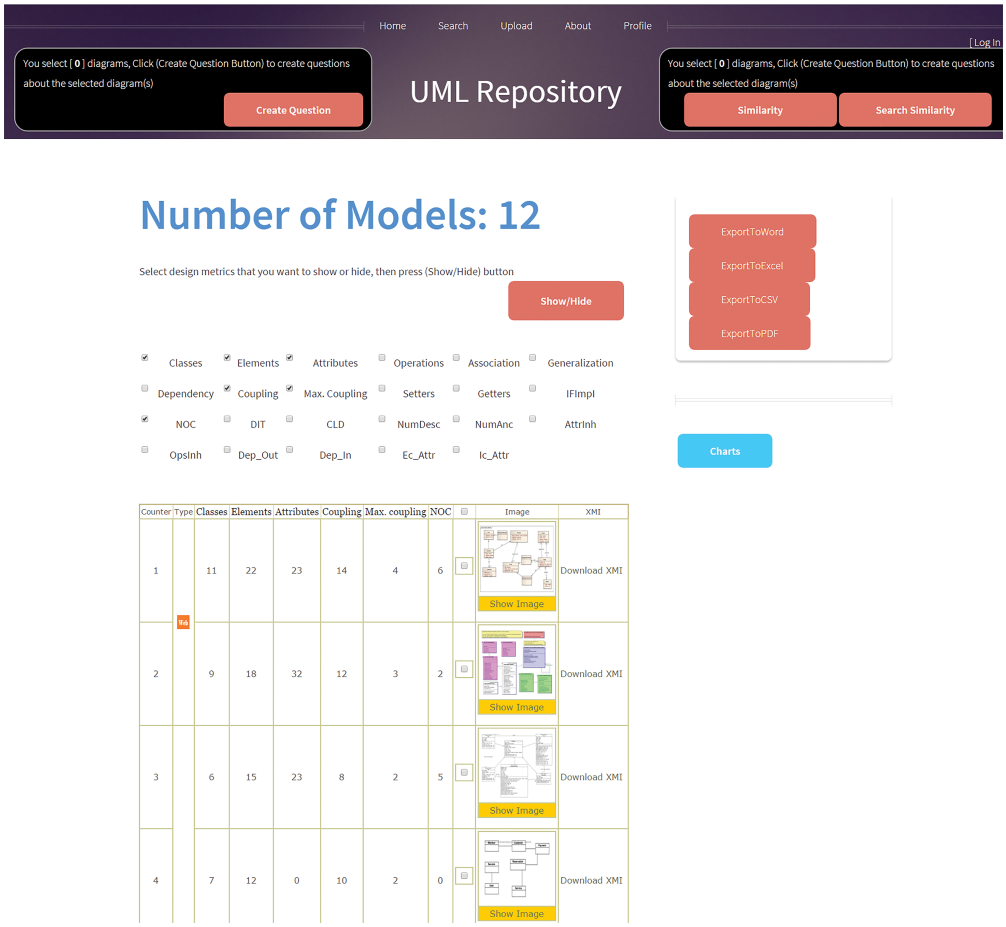

Figure 5.6: *Result page of searching for the class name "reservation"*

When users define question(s) about the first diagram, they can apply the same definition for all others diagrams selected or continue to define new questions. Users can define the answer type for each question. Answers can be:

- Multiple choices: can be in one of the three graphical user interface controls, which are radiobuttons, checkboxes and combobox.
- Text: can be in textbox control.

Figure 5.7 shows the question(s) page. We notice that the button (Apply Question(s) for All Diagrams) is visible only when the user select more than one diagram. At the end, users have a link to the questions that they define. This link can be shared with other user who can answer these questions.

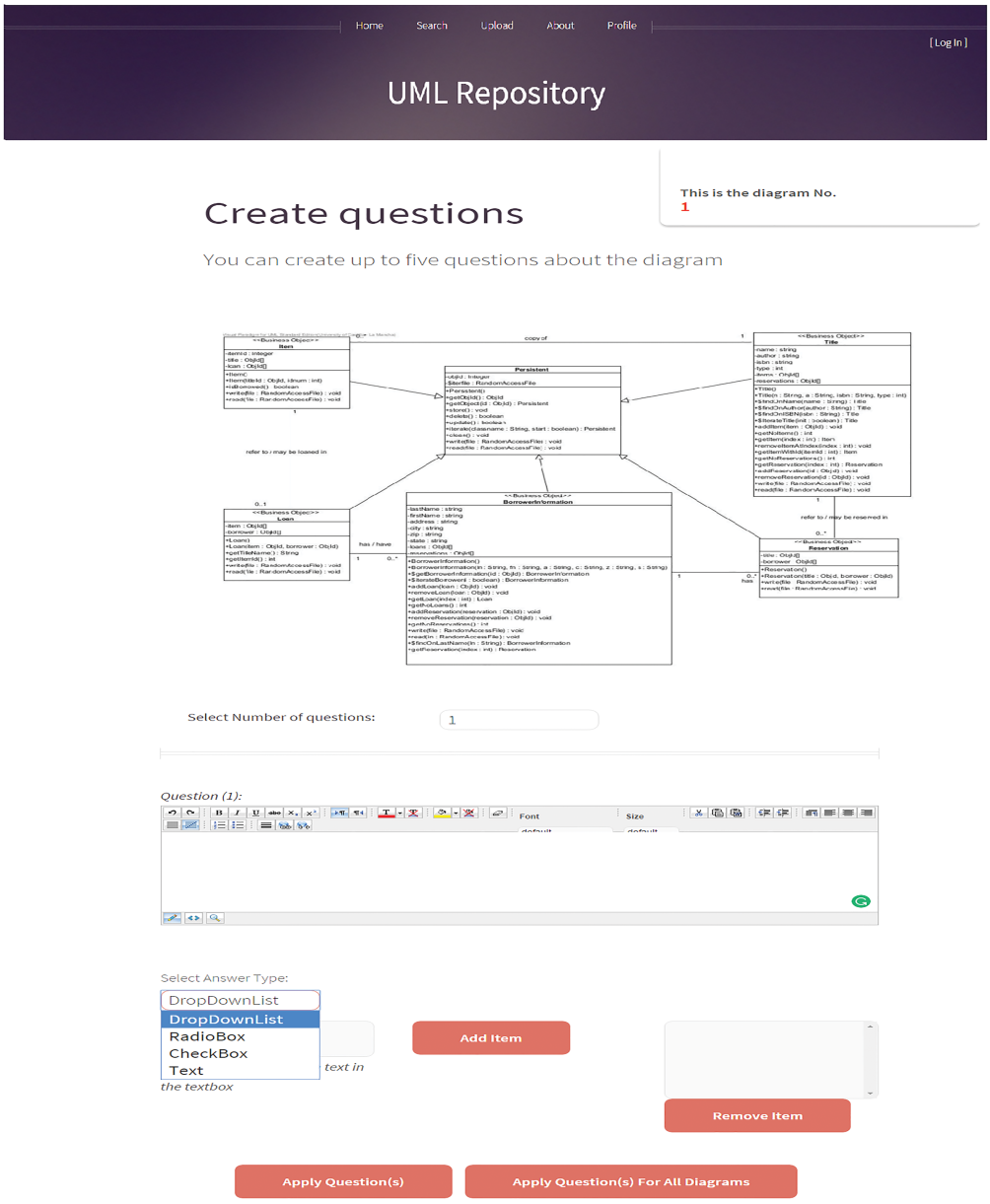

Figure 5.7: *Question(s) page*

5.6.2 Performing Experiments and Online Questions

Users can select many diagrams and create many questions about each one. Consequently, users can define experiments such as making a questionnaire asking about

some quality attributes of the selected diagrams and share the link of the questions with novices and experts. Another experiment is making a comparison between diagram features such as number of operations and number of coupling that makes diagrams more complex. Users can view their questions on their profile pages along with all the answers.

In addition, there are many diagrams in the repository that are related to experiments performed previously by other researchers. Hence, it becomes easy to replicate their experiments as the dataset is already available. Teachers can make online exams or quizzes and ask students to answer the quiz online. Teachers can distinguish students based on their username or from their comments where they can mention their IDs.

There are many online tools that offer online quizzes, but our repository has a collection of UML diagrams that are not available in these online tools. After users have made a selection of the diagrams, they can assemble these diagrams in a series of questions (questionnaire) that can be used in an experiment or an exam. The system then creates a link to this questionnaire, and the users can share this link with others.

5.6.3 Similarity Between Diagrams

The repository has two features that support a similarity measure between diagrams:

1. Select two diagrams and compute the similarity between them.

2. Select one diagram and search in the repository for diagrams that are similar to the given diagrams with respect to one or more design metrics within some threshold. The user interface offers a slider that sets the threshold value for each design metric.

Figure 5.8 shows the similarity page for two diagrams. This figure shows that the similarity-result-page contains a similarity table and a radar chart for design metrics value.

We measure the similarity between each design metrics between models using the equation 5.1.

$$\text{Similarity}(\text{Model}_1\text{Metric}_\alpha, \text{Model}_2\text{Metric}_\alpha) = \frac{Min(Model_1Metric_\alpha, Model_2Metric_\alpha)}{Max(Model_1Metric_\alpha, Model_2Metric_\alpha)} \quad (5.1)$$

5.6.4 Visualize Search Result

Our system can create graphs based on the metrics that belong to the set of query answers. We can classify repository-visualizations into three categories:

1. *Occurrences*: the system can show column and bar frequency distributions – as histogram charts - for the distribution of a design metric. Users also can switch

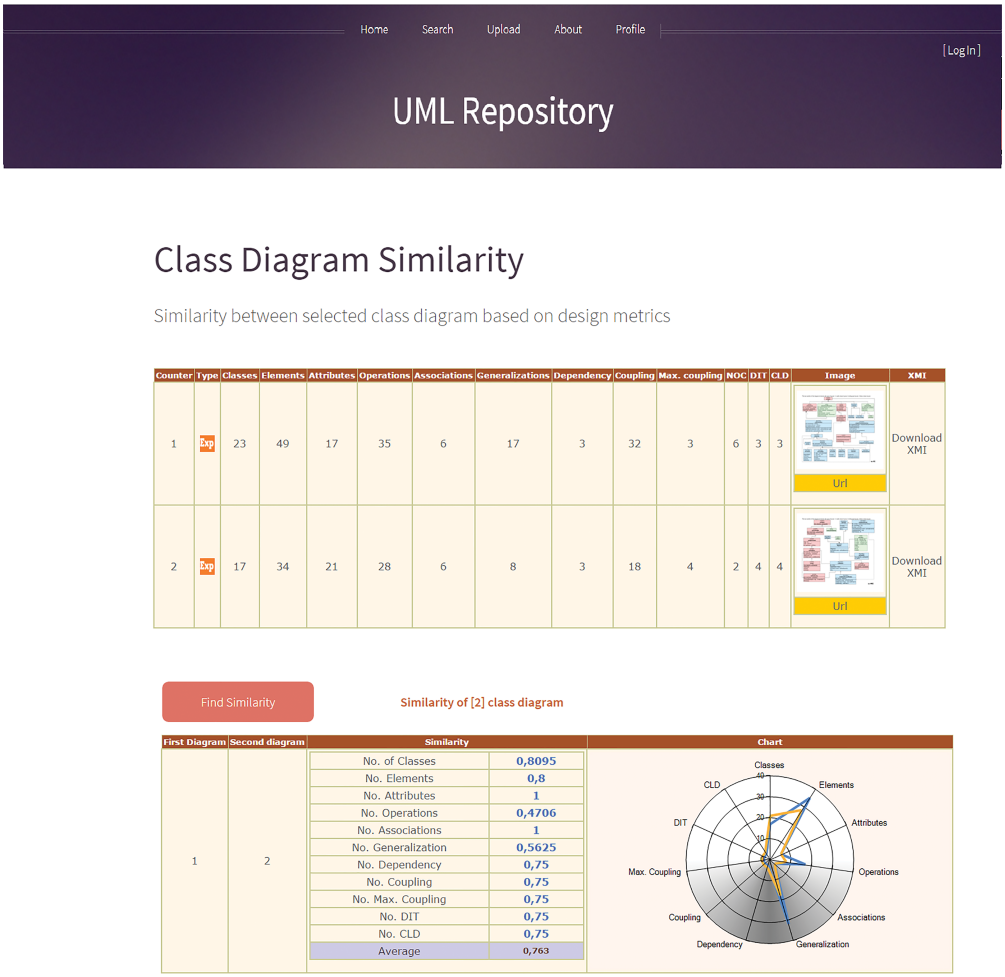

Figure 5.8: *Similarity between two class diagrams*

between a *column chart* and *bar chart*. In addition, there are many charts options such as column width, the style of the column and visualize chart in 3D mode. Figure 5.9 shows an example of the distribution of classes.

2. *Relations*: the system can show line, fast line and point line charts for relations between different design metrics. For example, the relation between the number of operations and the number of coupling. Figure 5.10 shows an example of the relation between the number of operations and the number of coupling. The system also can sort the results in ascending and descending order. Users can switch between line, fast line and point line charts. They can change border

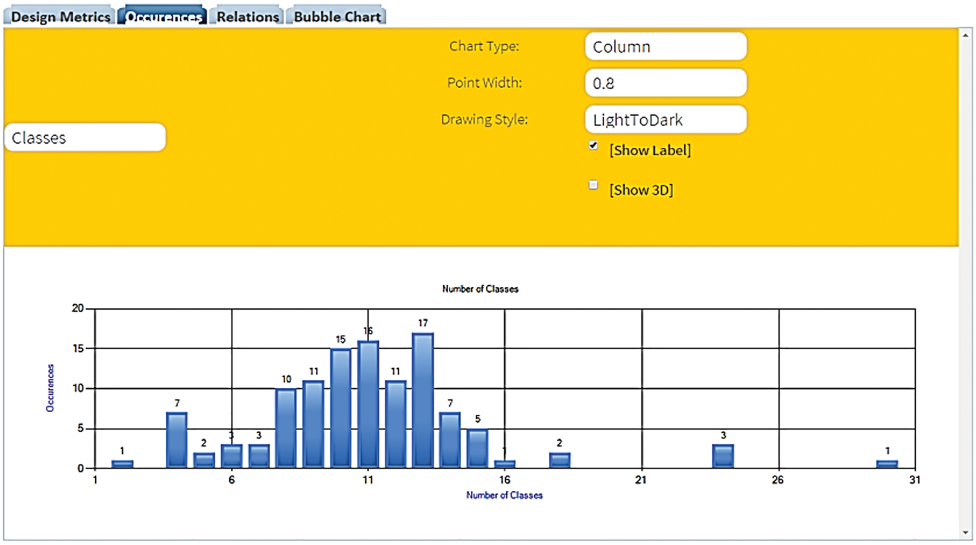

Figure 5.9: *Similarity between two class diagrams*

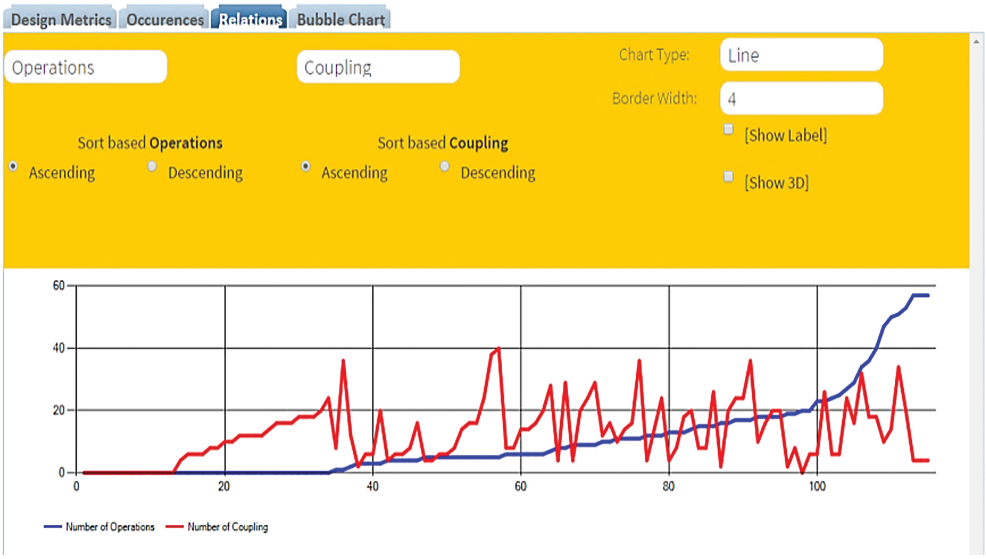

Figure 5.10: *Relation between operation and coupling*

width of the lines and show the chart in 3D mode.

3. *Bubble chart*: The system can show bubble charts of relations between design metrics. One advantage of bubble charts is that they show the size (in our case

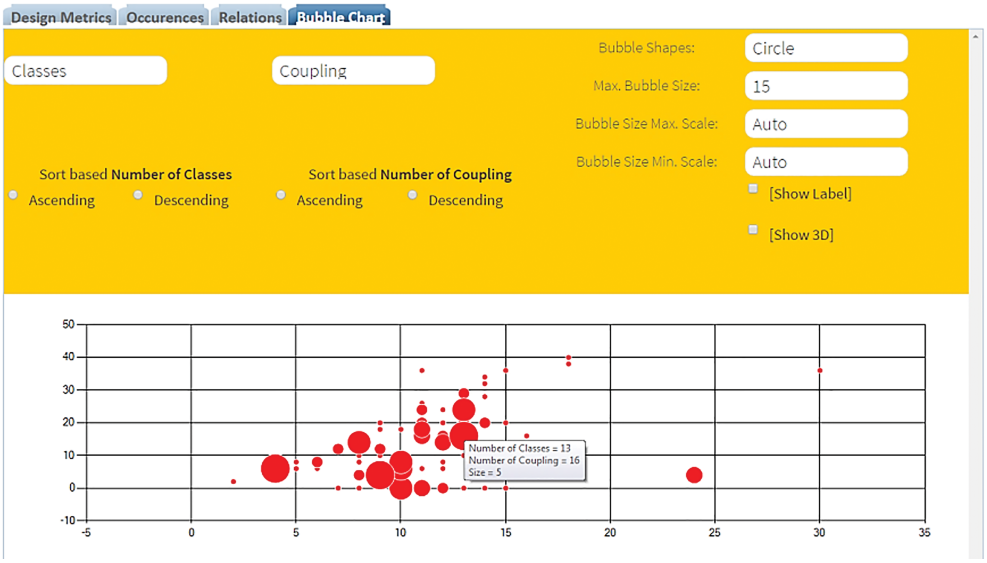

Figure 5.11: *Bubble chart for the relation between classes and coupling*

number occurrences) of each relation. Figure 5.11 shows a bubble chart example. When users hover the mouse over a bubble, more information of that bubble pops up including the value of the selected item on the X-axis, the Y-axes and the size of the bubble. Users can switch between several bubble shapes: circle, square, diamond, triangle, cross and star.

5.6.5 Uploading Models

Users can upload images of UML diagrams, XMI files and software documents as a project to the repository. The repository group publishes uploaded documents after manual examination. However, models uploaded by active users are published directly. When users upload images or XMI files, they can input tags and comments for their uploaded files. For uploading a project software documents, they can enter a project name, URL of the project, tags, a check box whether the project is related to an experiment and comments about the project. Figure 5.12 shows the uploading project documents page.

5.7 Conclusion and Future Work

In this chapter, we proposed our repository for UML models. The repository contains UML class diagrams images that are collected from the internet, literature, collaboration with other universities and students. The repository also contains the images' URLs and

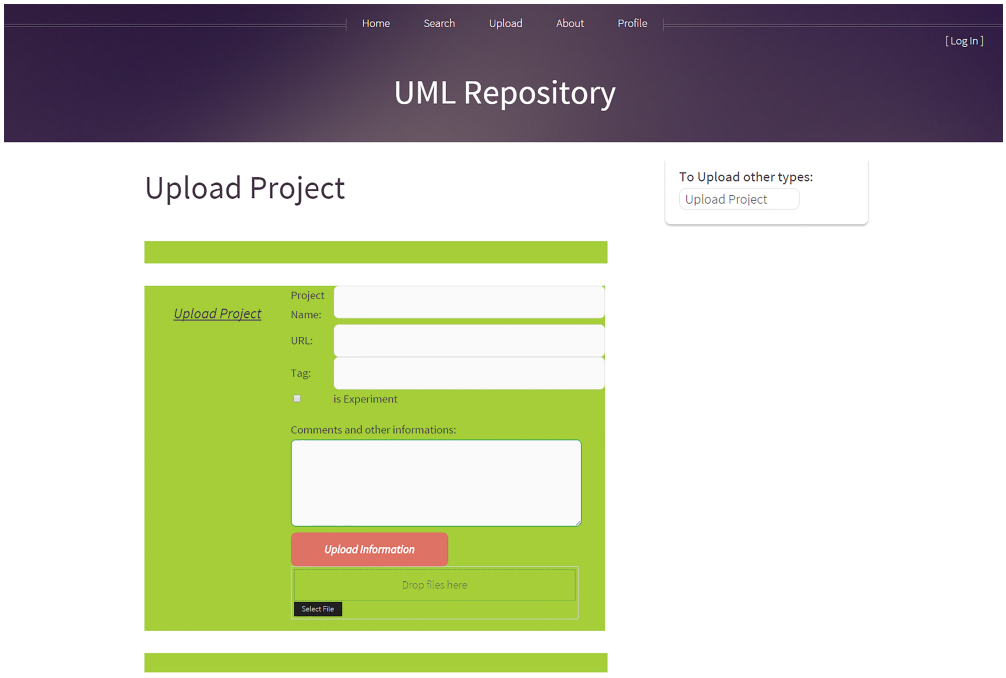

Figure 5.12: *Upload Project form*

the corresponding XMI files that are generated via Img2UML tool. A web-based user interface is available and the repository is open accessible. The goal of the repository is to be a basis for UML models that can be used and shared across empirical studies. We present our collecting approach, the repository schema and the repository services.

For future work, we plan to enrich the repository with other UML diagrams such as sequence diagrams and use case diagrams. In addition, we plan to integrate a UML-editor with the repository, to store user activities as well as their models. We believe that we can find some patterns in user behaviors while they create their models. These patterns can be matched with quality of their models, which could give us some guidelines on the best and bad practice in modeling.

Chapter 6

UML Repository As Benchmark for Quality Analysis

In this chapter, we illustrate some analysis that based on the UML Repository as a benchmark for quality analysis. First, we show common characteristics of class diagrams in the repository. In addition, we show the relation between models size and coupling metric. Then we describe our study of the relation between anti-patterns in design models and source code. We show the impact of the quality of the design models measured by anti-patterns on the quality of the source code measured by numbers of changes and faults.

This chapter is based on the following publications:

- Bilal Karasneh, Michel R. V. Chaudron, Foutse Khomh and Yann-Gaël Guéhéneuc. **Studying the Relation between Anti-patterns in Models and in Source Code.** *In Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering, Osaka, Japan. 2016.*
- Bilal Karasneh, Michel R. V. Chaudron. **Online Img2UML Repository: An Online Repository for UML Models.** *In Proceedings of the 3rd International Workshop on Experiences and Empirical Studies in Software Modeling (EESS-MOD@MODELS 2013), pages 61-66, Miami, USA. 2013.*

In chapter two, we talked about software quality models, and showed how to measure software quality. In this chapter, we spot some characteristics of class diagrams in the repository based on their design metrics. We think these characteristics are important to assess the quality of class diagrams, because it helps to find common patterns in class diagrams, and measures these patterns in terms of quality. Furthermore, we present our study of the relation between anti-patterns in the design and quality of source code based on number of changes and faults.

6.1 Common Characteristics of Class Diagrams

Finding out common characteristics of the designs provide a way to measure their quality. The repository contains a big collection of class diagrams, which are related to different application domains and created by different designers. Thus, it can be considered as a good place to apply empirical studies. We analyze the size of class diagrams and the relation between size and maximum coupling. These metrics influence the complexity of class diagrams. Based on that analysis, we aim to find common patterns. We notice that coupling includes three relationships: association, composition and aggregation.

6.1.1 The Size of Class Diagrams

In this subsection, we describe the size of the diagrams in the repository. Figures 6.1 and 6.2 show a boxplot of class diagrams size and its distribution in the repository respectively. Table 6.1 provides descriptive details of figure 6.1. We see that the maximum number of classes is relatively small. We see that it is not common to find a class diagram with a big size - The median number of classes is nine.

Table 6.1: *Descriptive statistics of the size of class diagrams in the repository*

	No. Diagrams	Median	Quartile 25%	Quartile 75%
Diagrams size	810	9	5	13

6.1.2 Maximum Coupling

We believe that maximum coupling in class diagrams can be an important indicator of the complexity of class diagrams. So if there is at least one class in a class diagram that show a high coupling, this determines the complexity of that class diagram. Figure 6.3 shows the boxplot of the maximum coupling of the class diagrams in the repository. Table 6.2 shows the descriptive statistics of Figure 6.3.

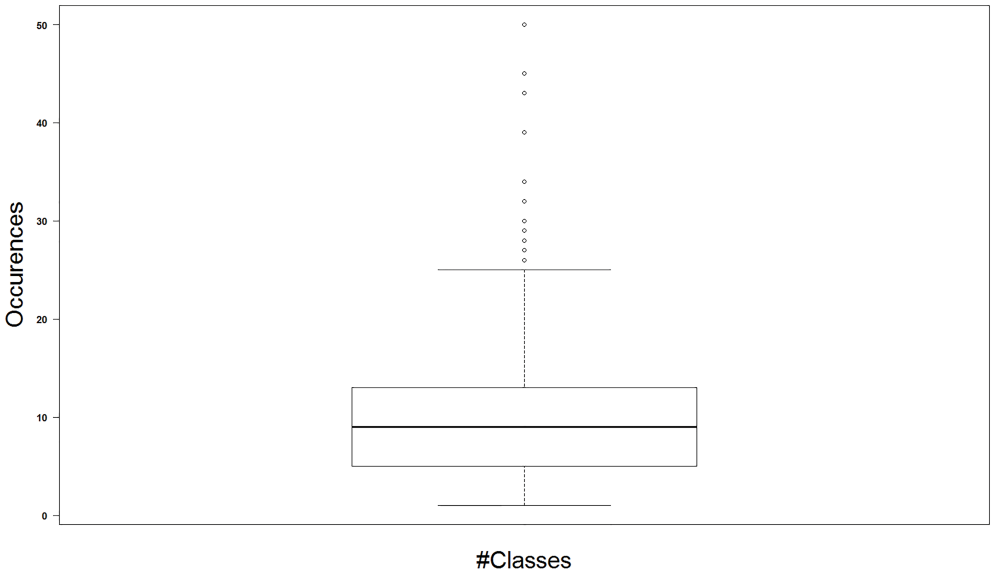

Figure 6.1: *Size of class diagrams in the repository*

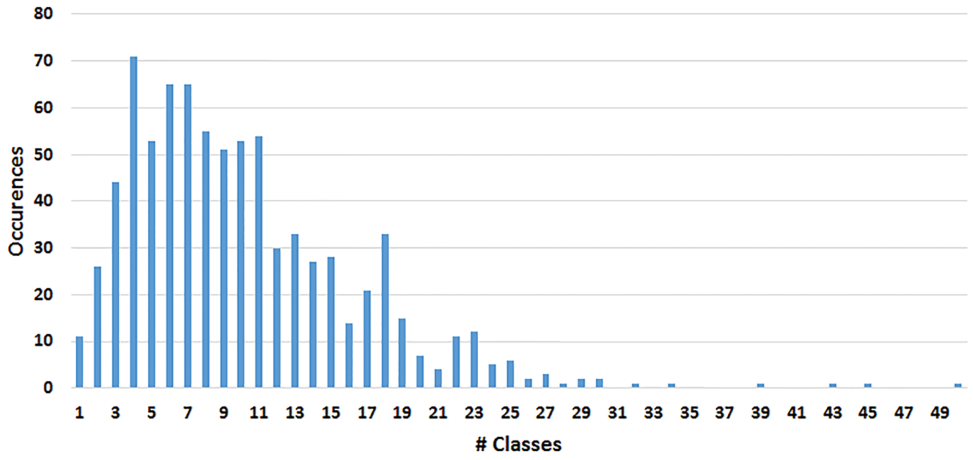

Figure 6.2: *Distribution of class diagrams size in the repository*

6.1.3 The Relation between Class Size and Max. Coupling

It is interesting to study the relation between the size of class diagrams and the maximum coupling that they indicate. Figure 6.4 shows a Bubble chart for diagrams size and their maximum coupling. Figure 6.4 shows that there is a positive correlation

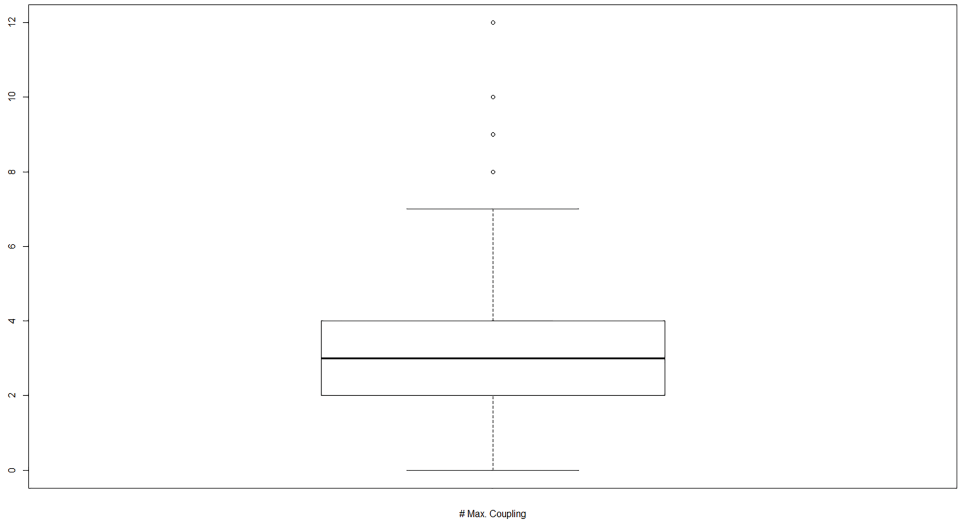

Figure 6.3: *Maximum coupling for diagrams in the UML Repository*

Table 6.2: *Descriptive statistics of Max. coupling in class diagrams in the repository*

	No. Diagrams	Median	Quartile 25%	Quartile 75%
Diagrams size	810	3	2	4

between diagrams size and maximum coupling. It shows that when the diagram size is high, the maximum coupling also will be high, which it makes diagrams more complex. In Figure 6.4, the trendline also shows the coefficient of determination (R2 = 0.07). A correlation of 0.10 is seen as a weak relationship, 0.30 as moderate, and 0.50 as strong relationship [80]. This interpretation is fit in our data because it is randomly collected, and the behavior, skill, and experience of the designer are different. The Pearson correlation that we calculated between the diagrams size, and the maximum coupling is significant ($p < 0.01$) and it is a moderate relationship ($r = 0.46$).

6.1.4 Discussion

Our study is a preliminary study of the size of class diagrams and maximum coupling. The size of the diagrams is relatively small, which it means that it is not common to have big diagram size. The maximum coupling is small, which could be considered as a hint of the complexity of class diagrams. The relation between class diagrams size and maximum coupling is moderate. However, this moderate relationship shows that the complexity of class diagrams increases when the size of class diagrams increases. We

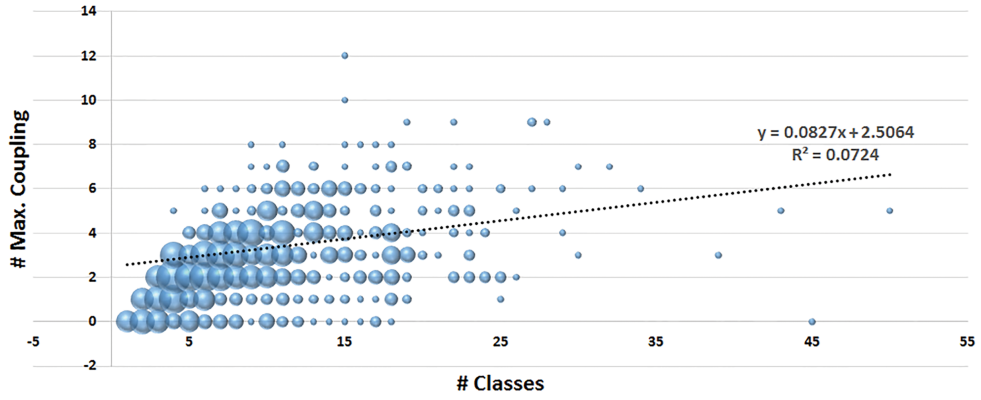

Figure 6.4: *Relation between Diagrams size and Max. coupling*

notice that 11% of class diagrams have a maximum coupling of zero. These diagrams do not contain any association, but they do inheritance and dependency relationships.

6.2 Studying the Relation between Design Quality and Source Code

Many software engineers focus on the quality of source code, and they do not give enough attention to the quality of the design. There are few studies about the contribution of software design on software development, because it is challenging to study.

In this section, we study the impact of software design on the source code. We investigated two open source software projects that have a design, different released versions of the source code, and we can access their changes and bugs among the different released versions. Then, we study the effect of anti-patterns in design on the quality of the source code measured by number of changes and number of bugs. For achieving this, we did two studies:

- First, we investigate seven open source software projects that have design and made a comparison between occurrences of anti-patterns in the design and the source code.

- Second, we measured the relation between anti-patterns in design and the quality of the source code based on changes and bugs. We use the same two open source projects that we used in the first experiment.

Table 6.3: *Descriptive statistics of Max. coupling in class diagrams in the repository*

Project Name	Modeled classes	Not-Modeled classes	Total
ArgoUML	88	2731	2819
Wro4j	197	876	1073

6.2.1 Relation between Software Design and Source Code

We conducted an experiment with two big open-source software projects to measure the quality of classes that appear both in the design and the source code, and classes that appear only in the source code. We selected ArgoUML and Wro4j because we can access to the class changes and bugs reports for all versions of the software. More information about ArgoUML and Wro4j is in Table 6.8. For ArgoUML, we use nine different released versions and for Wro4j, we use six different released versions.

6.2.1.1 Experiment Design

We categorized classes in the source code into two categories:

1. Modeled classes, which are classes that appear in the design and source code.
2. Not-Modeled classes, which appear in source code only.

Then we used Mann-Whitney test to compare the means of the number of changes and the number of bugs between both categories. We used Mann-Whitney test because it matches the precondition in both cases.

The number of classes in the design is small as it is usual in open-source software projects. We collected the corresponding of the classes that appear in the design and code. Therefore, the Modeled classes are the classes that appear in the design and the code, and their corresponding in the code, and the Not-Modeled classes are the classes that are only in the code. Table 6.3 shows both categories for both the ArgoUML and Wro4j projects. We use IntelliJ IDEA[1] to find the corresponding classes in the code.

6.2.1.2 Results

Tables 6.4 and 6.5 show the mean of changes and bugs for both Modeled classes and Not Modeled classes in both ArguUML and Wro4j, respectively. In ArgoUML, the result of the Mann-Whitney test shows that there is a significant difference between Modeled classes and Not-Modeled classes in both terms of changes and bugs, where p-value = 0.000 and p-value = 0.007 respectively.

In Wro4j, the results are the same, where the result of the Mann-Whitney test shows that there is a significant difference between Modeled classes and Not-Modeled classes

[1] https://www.jetbrains.com/idea/

Table 6.4: *Means of classes Changes in ArgoUML and Wro4j*

	Project Name	Categories	Mean
Changes	ArgoUML	Modeled classes	22
		Not-Modeled classes	7.78
	Wro4j	Modeled classes	15.17
		Not-Modeled classes	6.56

Table 6.5: *Means of classes faults in ArgoUML and Wro4j*

	Project Name	Categories	Mean
Bugs	ArgoUML	Modeled classes	0.82
		Not-Modeled classes	0.42
	Wro4j	Modeled classes	2.91
		Not-Modeled classes	1.04

in both numbers of changes and bugs, where p-value = 0.000 and p-value = 0.000, respectively.

Next, we show the relation between the number of changes and two code metrics, Line of Code (LOC) and average Cyclomatic Complexity (AvgCyc). Table 6.6 shows the correlations between number of changes in ArgoUML and Wro4j with LOC and AvgCyc. The number of changes in Modeled classes has significantly higher correlations with LOC than Non-modeled classes in both ArgoUML and Wro4j: Modeled classes that have higher LOC have more changes. Table 6.7 shows the correlation between faults, LOC and AvgCyc: the number of faults in Modeled classes have significantly higher correlations with LOC than Non-modeled classes in both ArgoUML and Wro4j. More faults exist in Modeled classes that have higher LOC. AvgCyc does not have a correlation with Modeled classes or Non-modeled classes.

6.2.1.3 Results Discussion

The result shows that there is a significant difference between Modeled classes and Non-Modeled classes categories in both cases changes and bugs. The means of changes and bugs of Modeled classes are higher than Non-Modeled classes, which mean the Modeled classes have more changes and bugs in both ArgoUML and Wro4j among different released versions. We explain this result as Modeled classes important and could have the main functionality of the system, so it has more changes during different released versions. We explain the faults by the positive relation between the number of changes and number of faults [81]. Therefore, classes in the design have more changes, this tend to have more faults. Indeed, the implementation should follow the design, which results in developers transferring problems from the design to the source code. We notice that, problems in the design may cause and propagate more problems in the

Table 6.6: *Correlation between Changes, LOC and AvgCyc*

	Project Name	Categories	Correlation		R^2	Formulas
			AvgCyc	LOC		
Changes	ArgoUML	Modeled classes	0.30	0.74	0.54	Y=0.727+0.018(LOC)
		Not-Modeled classes	0.21	0.43	0.19	Y=1.352+0.004(LOC) +0.092(AvgCyc)
	Wro4j	Modeled classes	0.06	0.62	0.39	Y=-1.937+0.315(L0C)
		Not-Modeled classes	0.00	0.38	0.17	Y=6.191+0.104(LOC)(AvgCyc)

Table 6.7: *Correlation between Changes, LOC and AvgCyc*

	Project Name	Categories	Correlation		R^2	Formulas
			AvgCyc	LOC		
Bugs	ArgoUML	Modeled classes	0.35	0.53	0.32	Y=-0.54+0.001(LOC) +0.041(AvgCyc)
		Not-Modeled classes	0.13	0.26	0.07	Y=0.07+0.000(LOC)
	Wro4j	Modeled classes	0.02	0.6	0.35	Y=-0.266+0.060(LOC)
		Not-Modeled classes	-0.01	0.40	0.18	Y=0.953+0.026(LOC) -0.417(AvgCyc)

source code because in the code more information and functions should be added.

6.2.2 Effects of Anti-patterns on Software Quality

There are very few studies on the origins of the occurrences of anti-patterns in the source code. Knowing where and when anti-patterns are introduced could help software designers and developers improve the quality of the software systems. In this section, we explore the origins of anti-patterns by tracing them back to design models. We conduct our study using the models selected randomly from the UML Repository, which is a unique repository containing pairs of architects and designers' models (UML class diagrams) linked to the corresponding source code, when available. For each system, we analyze both its UML design models and its source code.

Since their inception in software engineering, design patterns (i.e., reusable solutions to recurring design problems) [82] and anti-patterns (i.e., poor solutions to design and implementation problems) [83] have been the subject of many research works. This research works focused on design patterns specification [84], detection [85], and on the analyzes of their impact and life-cycle. Tufano et al. [86], Vaucher et al. [87], and Chatzigeorgiou and Manakos [88], investigated the evolution of anti-patterns in software systems, and observed that anti-patterns are not necessarily only introduced in the source code during maintenance and evolution activities. They reported that many classes are "born" anti-patterns. Following on this observation, we set out to investigate whether design models produced by architects and designers before the implementation contains anti-patterns. In addition, we see if theses anti-patterns translate in the source code, i.e., if these anti-patterns concern the same classes in design models and the source code implementing these models.

On the one hand, as long as the code follows the decisions embedded in the models, the same patterns/anti-patterns from the models should appear in the source code. On

the other hand, if we can use the right patterns and follow the right design decisions early on in the development cycle, we could prevent the occurrence of anti-patterns in the code.

6.2.2.1 Related Work

There are many research works on the definition, specification, detection, correction, and analysis of the life-cycles of design patterns and anti-patterns. Because we focus on anti-patterns, we describe here three works that (1) showed that anti-patterns do impact negatively class change- and fault-proneness, (2) studied the introduction and removal of some anti-patterns qualitatively, and (3) reported four lessons on their life-cycles. We describe the specification and detection of anti-patterns in Sections 6.2.2.3 and 6.2.2.4.

Khomh et al. [89] investigated the impact of anti-patterns on classes in object-oriented systems by studying the relation between anti-patterns and class change- and fault-proneness. They showed that in 50 out of 54 releases of the four analyzed systems, classes participating in anti-patterns are more change and fault-prone than others.

Vaucher et al. [87] studied the "God class" anti-patterns, which describes large classes that "know too much or do too much". The literature postulated that God classes are created by accident as functionalities are incrementally added by developers to central classes over the course of their evolution, Vaucher et al. observed that, in some systems, God classes are created by design, as the best solution to a particular problem, for example, when a functionality is not easily decomposable or when there exist strong requirements on efficiency or performance. They studied the life-cycles of God classes in the source code of Eclipse JDT and Xerces; investigating how they arise, how prevalent they are, and whether they remain as the system evolves over time. They distinguished between those classes that are God classes by design from those that occurred by accident in the implementation. They concluded that some God classes are created by design but most are the result of a decay of the systems. They propose that developers use detection techniques and refactoring to track and prevent anti-patterns in their systems.

Following this previous work, Tufano et al. [86] studied the life-cycle of five anti-patterns in Android, Apache, and Eclipse and drew the following lessons: (1) classes often play roles in anti-patterns from their inception in the systems, (2) the metric values of the classes that started to play some roles in some anti-patterns only during the evolution of the systems have specific trends, (3) refactoring operations, in addition to other changes, may lead to the introduction of anti-patterns, and (4) time pressure is the main cause of the introduction of anti-patterns. With these lessons, in the particular lesson (1), they confirmed the hypothesis of this present study that anti-patterns are not necessarily the results of developers' lack of times/skills but could be due to the very designs of the systems.

Chatzigeorgiou and Manakos [30], who investigated the evolution of anti-patterns

Table 6.8: *Studied Software Systems*

Project Name	Descriptions	URLs
ArgoUML	An open source UML modeling tool	`http://argouml.sourceforge.net`
Annoyme	Adds beautiful typewriter sounds to Desktop keyboards	`https://github.com/dedeibel/annoyme`
JGAP	Package of Genetic Algorithm and Genetic Programming	`http://jgap.sourceforge.net`
Mars_Simulation	Project to create a simulation of future settlements on Mars	`http://mars-sim.sourceforge.net`
Msv_Poker	Poker Game (poker server and poker client)	`https://github.com/mihhailnovik/msvPoker`
Neuroph	Lightweight Java neural network framework to develop network architecture	`http://neuroph.sourceforge.net`
Wro4j	Web resource optimizer for Java	`http://code.google.com/p/wro4j`

in object-oriented systems reported that anti-patterns tend to linger in systems for multiple releases.

All these studies considered occurrences of anti-patterns in the source code of the studied systems (and their revisions) or design models reverse-engineered from their source code. They did not study the prevalence of anti-patterns in design models created before (and–or during) development in comparison to that in the source code implementing these design models. Our following study aims at confirming the observations that, in some designs, anti-patterns are present from the very beginning of the inception of the systems. In addition, these anti-patterns in the design have an impact on the implementation.

6.2.2.2 Experiment Design

The seven open-source systems selected from UML Repository are available from Github, SourceForge, and Google Code. The studied software systems are in Table 6.8. We notice that it is much easier to use UML Repository for finding such case studies that contain UML design with source code. It is hard to inside code repositories to find software projects that contain UML design. We conduct this study using both architects' and designers' models of software systems and the implementation of these models as source code.

6.2.2.3 Anti-patterns Identification

We use the Ptidej[2] tool suite, which implements the anti-pattern detection approach DECOR (Defect dEtection for CORrection) [85], to identify occurrences of anti-patterns in both models and source code. DECOR is an approach based on the automatic generation of detection algorithms from rule cards. It converts anti-patterns descriptions automatically into detection algorithms and identifies the occurrences of these anti-patterns in UML class diagrams and the source code of systems.

We apply DECOR in three steps: first, we reuse/define a rule card describing an anti-pattern through a domain analysis of the literature [84]. From the rule card, we generate a detection algorithm. Finally, we apply the detection algorithm on models of systems to detect the different occurrences of the anti-pattern in these systems. DECOR has appropriate performance, precision, and recall for our study.

DECOR can be applied to any object-oriented system through the use of the PADL [90] meta-model and POM framework [91]. PADL describes the structure of systems and a subset of their behavior, i.e., classes and their relationships. POM is a PADL-based framework that implements more than 60 structural metrics. We apply DECOR on models obtained either from the class diagrams available in the repository, by parsing the corresponding XMI files, or by parsing the corresponding C++ and Java source code.

6.2.2.4 Anti-patterns Specification

Concretely, we detect the occurrences of four anti-patterns which are: (1) Complex class, (2) Large class, (3) Lazy class, and (4) LongMethod. We are using the metrics available in Ptidej for both class diagrams and source codes.

Essentially, we specify the Lazy Class in terms of the number of methods defined in the classes. We define the Complex Class as the number of methods and the relationships among classes: a class that defines many methods and that has many relationships (in or out) is inherently complex. Long Method can only be computed on classes from the source code because we need the number of statements in the methods. Finally, we specify Large Class as the "opposite" of a Lazy Class, in terms of the number of methods in the classes. The details of the specification and detection of the anti-patterns are outside the scope of this chapter because we reuse the specifications and detection algorithms used in previous work and detailed in the presentation of DECOR to which we refer the reader [84].

6.2.2.5 Results

We now report the results of detecting anti-patterns in models and source code of seven systems from the UML Repository using Ptidej. First, we show some analysis of

[2]http://www.ptidej.net

Table 6.9: *Summary of number of Classes in class diagrams versus in source code*

Project Name	# Classes in class diagrams	# Classes in source code	Proportion of classes
Annoyme	17	59	0.29
ArgoUML	51	1722	0.03
JGAP	19	191	0.1
Mars_Simulation	32	953	0.03
Msv_Poker	22	55	0.4
Neuroph	26	179	0.15
Wro4j	28	598	0.05

the numbers of classes in the models and their source code. Then, we summarize the anti-patterns detected in both models and their source code.

Table 6.9 shows an overview of classes in the models and source code, which proportions are based on the equation 6.1:

$$\text{Proportion of classes} = \frac{\text{No. of classes in the Design}}{\text{No. classes in the source code}} \tag{6.1}$$

The numbers of classes in the source code are higher than in the models, but we found that some classes in the models were missing in the source code. This is also expected because the models, which are conceptual models, are often refined by developers during implementation. However, these refinements are not always documented back in the models. In Table 6.10, we show the proportions of classes that exist in models and source code. The proportions in Table 6.10 are measured using the following equations 6.2 and 6.3:

$$\text{C.C.C. to the Design} = \frac{\text{No. classes exist in both Design and source code}}{\text{Number of class in Design}} \tag{6.2}$$

$$\text{C.C.C.totheImplementation} = \frac{\text{No. of classes exist in both Design and source code}}{\text{Number of class in source code}} \tag{6.3}$$

C.C.C. = Common Classes Compared
Common Classes = Classes exist in class diagrams and the implementation

The majority of classes contained in the models are also present in the source code of the systems. However, classes in the class diagrams represent only a fraction of the total numbers of classes contained in the source code. Nevertheless, next we show that

Table 6.10: *Proportion of classes that exist in both class diagrams and source code*

Project Name	# Common Classes	Common classes compared to the Design	Common classes compared to the Implementation
Annoyme	14	0.82	0.24
ArgoUML	44	0.86	0.03
JGAP	18	0.95	0.09
Mars_Simulation	29	0.91	0.03
Msv_Poker	13	0.59	0.24
Neuroph	24	0.92	0.13
Wro4j	23	0.82	0.04

anti-patterns appear in class diagrams during the design phase are transferred to the implementation.

We can detect three anti-patterns in the class diagrams: Complex Class, Large Class, and Lazy class. We can only detect LongMethod in the source code because class diagrams are abstract representations of the systems, and they contain only method signatures without the implementation details needed to compute the lengths of the methods. In Table 6.11, we report the numbers of anti-patterns that we found in the class diagrams and source code of the studied systems. We calculate the proportion of classes that play the same roles in the same anti-patterns in class diagrams and the source code based on Equation 6.4:

$$\text{Proportion of classes} = \frac{\text{No. S.AP.in.D.and.I}}{\text{No. Anti} - \text{patterns in class diagrams}} \tag{6.4}$$

S.AP.in.D.and.I = Same anti-patterns in the same classes in Design and implementation

Table 6.12 shows the number of anti-patterns that exist in the same classes in class diagrams and the source code. We also show the proportion based on Equation 6.4. From Table 6.12 we show that there is a significant proportion of classes playing the same role in the same anti-patterns in the class diagrams, and the source code **(36%)**. We notice that some anti-patterns appear in class diagrams and the same classes in the source code have different anti-patterns. We relate this to a common mistake in both open-source and commercial software development that they update the source code and do not update the design.

Next, we focus on individual anti-patterns and occurrences of each one in class diagrams and source code.

Table 6.11: *Anti-patterns detection in both class diagrams and source code*

Project Name	# APs* in Design	# APs in the Implementation	# Long-Method APs in the source code	# Same APs in the same classes in Design and Implementation
Annoyme	10	16	0	5
ArgoUML	20	545	256	10
JGAP	14	252	130	5
Mars_Simulation	24	370	206	3
Msv_Poker	16	18	8	4
Neuroph	12	41	27	4
Wro4j	28	209	130	12

**APs = Anti-patterns*

Table 6.12: *Proportion of classes in class diagrams that transfer same anti-patterns to the source code*

Project Name	# APs* in Design	# Same APs in the same classes in Design and Implementation	Proportions of same classes have same APs in Design and Implementation
Annoyme	10	5	0.5
ArgoUML	20	10	0.5
JGAP	14	5	0.38
Mars_Simulation	24	3	0.12
Msv_Poker	16	4	0.25
Neuroph	12	4	0.33
Wro4j	28	12	0.43
Average			**0.36**

**APs = Anti-patterns*

6.2.2.6 Complex Class

Regarding the Complex Class anti-pattern, very few occurrences are found in class diagrams, which means that architects and designers' tend to avoid excessively complex classes. However, developers do not seem to follow the same care during implementation as we observe proliferations of occurrences of the Complex Class anti-pattern in the source code of ArgoUML, JGAP, Mars, and Wro4j. Figure 6.5 shows show the number of occurrences of Complex class anti-patterns detected in the class diagrams and source code. We explain this observation by two facts. On the one hand, models

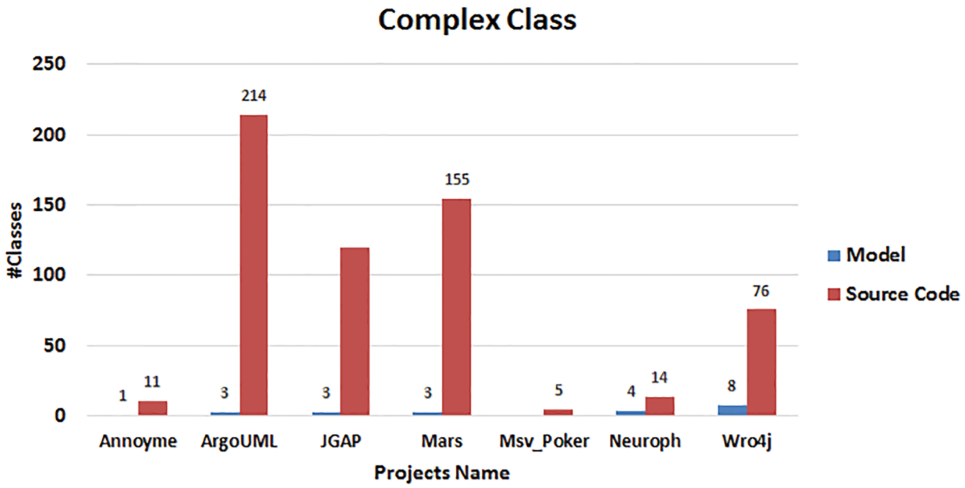

Figure 6.5: *Occurrences of the complex class anti-pattern in class diagrams and source code*

tend to be sketches of the actual implementation and, hence, do not contain all the details and complexity of the source code while the source code must, by its very definition, contain the actual algorithms, which may be intrinsically complex to implement. On the other hand, complex classes in source code tend to arise because of the lack of time for developers to research the best (i.e., simplest) implementation. Hence, it is our experience and observation that source code tends to be inherently more complex than necessary and, therefore, more complex that the models.

6.2.2.7 Large Class

Occurrences of the Large Class anti-pattern are absent from both models and source code, except for Wro4j, whose model contains five occurrences of the Large class anti-pattern, as shown in Figure 6.6. As reported by Vaucher et al. [87], Large Classes are sometimes present in systems because they are the best solution to some problems, for example when the problem is not easily decomposable. Such cases seem to be rare in models: only one system out of seven contains occurrences of the Large Class for the same reasons as mentioned above. So, modelers' focus on the essentials of classes, developers lack of time to introduce proper abstractions and, thus, their tendency to "grow" classes to implement new features.

6.2.2.8 Lazy Class

Lazy class, which is the most frequent anti-pattern among the four anti-patterns under study, is more prevalent in models than source code (see Figure 6.7. We explain this

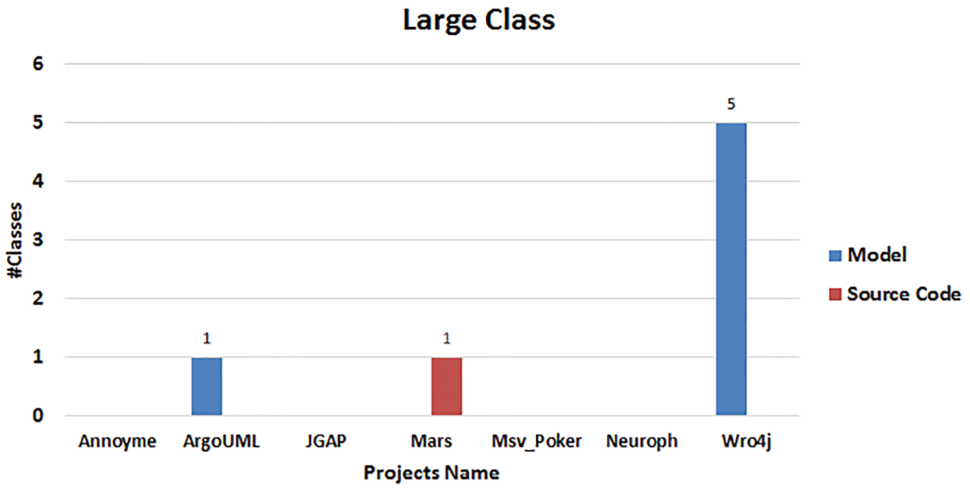

Figure 6.6: *Occurrences of the large class anti-pattern in class diagrams and source code*

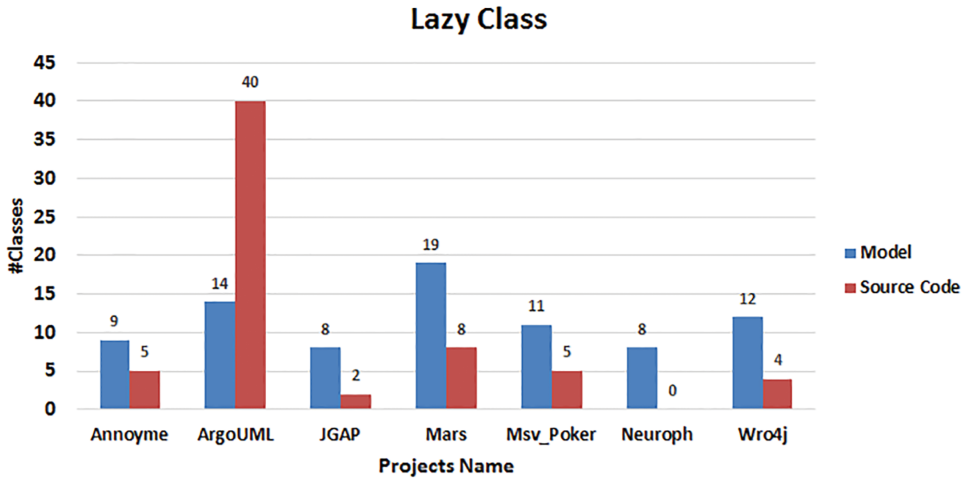

Figure 6.7: *Occurrences of the Lazy class anti-patterns in class diagrams and source code*

result by the fact that, during the design phase, developers try to anticipate future evolutions of the systems, which often lead to many abstract classes that do not contain necessarily enough behavior to justify their existence. These classes are considered Lazy classes by our detection technique and by definition of the anti-pattern. However, as Figure 6.7 shows, in all systems but ArgoUML, these excessive abstractions are corrected later by developers during the implementation of the systems.

6.2.2.9 LongMethod Class

Occurrences of the Long Method anti-pattern are also introduced in the source code in large number by developers. Again, we explain this observation and the difference between models and source code in two ways. First, models do not contain all the details necessary to identify Long Methods because of their very nature as sketches. Second, as previously mentioned, developers tend to implement features as fast as possible, under time pressure, and thus cannot take the time required to refactor their code and to avoid long methods.

6.2.2.10 Result Discussion

From the results presented in the results section, three of the four anti-patterns under study could be detected in class diagrams, i.e., during the design, which is considered an early stage of the software development life-cycle. We could not find occurrences of the Long Method anti-pattern in models because the detection of this anti-pattern is based on the numbers of Line of Code (LOC) of the methods, which is not available in class diagrams.

Table 6.10 shows that some classes contained in the models disappear in the source code, which can be considered two ways. First, having a class in a model and not having this class in the source code could be a design violation. For example, an architect or designer could have introduced a Facade between two subsystems, later the Facade is removed by developers for the sake of simplicity of implementation or performance of execution. Such a removal could yield to unintended accesses to some subsystems and also reduce information hiding.

However, having less information in models and not in source code such as classes, can also be the result of a lack of traceability in the project, because developers may have refined the model during implementation while failing to document the modifications and updating the models. Indeed, updates and changes to the source code without updating the models is a common practice observed in many software projects. Hence, our results confirm the software engineering lore that models are not synchronized with their source code by developers.

In addition, we observe that most of the classes that are in models and disappear in source code are Lazy Classes (see Figure 6.7, which confirms our intuition about developers refining the models because Lazy classes are the result of excessive abstractions. With a better knowledge of the system under development, developers may decide to remove some of the abstractions that result from architects' and designers' speculations about future evolutions of their systems. The average proportion of the same classes in the models and the source code that have same anti-patterns is 36%, which represents an important fraction of the total number of anti-patterns contained in the design. Hence, by acting early on these anti-patterns, architects, designers, and developers could improve the quality of their systems. Defects contained in design

models are known to be particularly expensive if they are not fixed quickly because classes in the models are the backbone of the source code and, in most models, are the most important classes in the source code. Thus, our results report and confirm that (1) classes in models may have anti-patterns, which propagate to the source code; and, (2) classes in both models and source code have the same anti-patterns in half the cases.

Also, following the broken windows theory [92], which states that a broken window may lead to a general degradation of the whole environment, we make the following argument. Similarly, we argue that when design problems are not fixed quickly, they tend to propagate in the system causing other problems. Therefore, it is important to track and fix design problems as early as possible in the development cycle. The results of this study show that software organizations can make use of anti-patterns detection tools like Ptidej during the design phase and track and fix anti-patterns in their software system as early as the design phase. Thus, anti-patterns detection tools will help prevent defects that could occur because of anti-patterns. Indeed, the refactoring of anti-patterns should be easier and less costly at modeling level than during implementation.

6.2.3 Effects of Anti-patterns in design on Software Changes and Faults

In this study, we focus on seven types of anti-patterns, Complex, Large, Lazy, Blob, ClassDataShouldBePrivate, RefusedParentBequest, and BaseClassShouldBeAbstract classes. We study the effect of anti-patterns of the classes appear in the design and the source code. We study the impact of anti-patterns appear in the design on the quality of the same classes in the source code measure based on the number of change- and fault-proneness the source code. Indeed, we use the Ptidej tool suite for detecting anti-patterns in the design (class diagrams).

6.2.3.1 Experiment Design

We categorized classes in the design into two categories:

- Anti-patterns category, which contains classes that have anti-patterns in design.
- No-Anti-patterns category, which contains classes that do not have anti-patterns in design.

Then we use the Mann-Whitney test to compare the means of the number of changes and the number of bugs between both categories.

Because the number of classes in designs is small as usual in open-source software projects, we did the same as in section 6.2.1.1 that we collected classes in the design and their mapped classes in the implementation. We use IntelliJ IDEA to find the mapped classes in the implementation to the classes in the design. Therefore, the Anti-patterns category becomes the classes that have anti-patterns in the design and their

Table 6.13: *Summary of classes used in the experiment*

Project Name	Anti-patterns Category	No-Anti-patterns Category	Total
ArgoUML	56	32	88
Wro4j	130	69	199

Table 6.14: *Means of classes changes in ArgoUML and Wro4j*

	Project Name	Categories	Mean
Changes	ArgoUML	Anti-pattens	30.8
		No-Anti-patterns	06.59
	Wro4j	Anti-pattens	16.6
		No-Anti-patterns	12.26

mapped classes in the implementation, and the No-Anti-patterns category contains the classes that do not have anti-patterns in the design and their mapped classes in the implementation. Table 6.13 shows both categories in both the ArgoUML and Wro4j projects. We take into account the number of changes and bugs occurred in different versions of both ArgoUML and Wro4j. We collected anti-patterns, changes and bugs for each class, then we entered the collected data into a database and made some queries for analyzing, filtering and organizing categories based on classes in the design (with their mapped classes) and occurrences of bugs and changes in the implementation of all versions.

6.2.3.2 Results

Tables 6.14 shows the mean of changes for both categories in the both ArguUML and Wro4j. Tables 6.15 shows the mean of bugs for both categories in the both ArguUML and Wro4j. We use the Mann-Whitney test because the test fits in our cases. The result of the Mann-Whitney test shows that there is a significant difference between the Anti-patterns category and the No-Anti-patterns category in ArgoUML in terms of changes and bugs, where p-value = 0.000 and p-value = 0.015 respectively.

In Wro4j, the results are the same, where the results of the Mann-Whitney test show that there is significant difference between the Anti-patterns category and the No-Anti-patterns category in both numbers of changes and bugs, where p-value = 0.000 and p-value = 0.004 respectively.

Table 6.16 and Table 6.17 show the correlations between numbers of changes and faults in Anti-patterns classes and No-anti-patterns classes with both LOC and AvgCyc. Table 6.16 shows that the numbers of changes have significantly higher correlations with LOC in No-anti-patterns classes than Anti-patterns classes in both ArgoUML and Wro4j: in systems with anti-patterns, size is not the only factor affecting change-proneness. The occurrence of anti-patterns also contributes to the occurrence of changes.

Table 6.15: *Means of classes faults in ArgoUML and Wro4j*

	Project Name	Categories	Mean
Bugs	ArgoUML	Anti-pattens	1.2
		No-Anti-patterns	0.15
	Wro4j	Anti-pattens	3.16
		No-Anti-patterns	2.38

Table 6.16: *Correlation between Changes, LOC and AvgCyc*

	Project Name	Categories	Correlation		R^2	Formulas
			AvgCyc	LOC		
Changes	ArgoUML	Anti-patterns classes	0.41	0.67	0.44	y=0.997+0.017
		No-anti-patterns classes	0.33	0.85	0.72	Y=0.149+0.028(LOC)
	Wro4j	Anti-patterns classes	0.06	0.59	0.35	Y=0.989+0.245(LOC)
		No-anti-patterns classes	0.02	0.7	0.48	Y=-0.698+0.465(LOC)

Table 6.17 shows the correlations between numbers of faults, LOC, and AvgCyc in ArgoUML and Wro4j: the numbers of faults in No-anti-patterns classes have significantly higher correlations with LOC than Anti-patterns classes in both ArgoUML and Wro4j. More faults occur in bigger No-antipatterns classes. For Anti-pattern classes, there is no strong correlation with LOC, which means that faults exist no matter the size of the classes. For AvgCyc, the correlations with changes is higher in Anti-patterns classes, which means that complex classes have more changes.

6.2.3.3 Discussion

The results show that there is a significant difference between Anti-patterns category and Non-Anti-patterns category. Therefore, and because of the mean of changes and mean of bugs of Anti-patterns category are bigger than No-Anti-patterns category in both ArgoUML and Wro4j, we conclude that the classes that have anti-patterns at design in ArgoUML and Wro4j, have more changes and bugs in the implementation.

From this experiment, we observe that classes that have antipatterns in the designs and corresponding classes in the source code of ArgoUML and Wro4j have more changes and faults in the implementation.

The broken windows theory [93] states that a broken window may lead to a general degradation of the whole environment and we argue that developers should solve these design problems before transferring them to the source code to reduce implementation and maintenance effort.

Similarly, we argue that when design problems are not fixed quickly, they tend to propagate in the system causing other problems. It is therefore, important to track and fix design problems as early as possible in the development cycle. The results of this study show that software organizations can make use of anti-patterns

Table 6.17: *Correlation between faults, LOC and AvgCyc*

	Project Name	Categories	Correlation		R^2	Formulas
			AvgCyc	LOC		
Bugs	ArgoUML	Anti-patterns classes	0.48	0.61	0.42	Y=-0.90+0.001(LOC) +0.054(AvgCyc)
		No-anti-patterns classes	0.35	0.81	0.65	Y=-0.072+0.001(LOC)
	Wro4j	Anti-patterns classes	-0.01	0.57	0.49	Y=0.377+0.046(LOC)
		No-anti-patterns classes	0.01	0.7	0.48	Y=-1.534+0.097(LOC)

detection tools like Ptidej during the design phase and track and fix anti-patterns in their software system as early as the design phase. Thus, anti-patterns detection tools will help prevent defects that could occur because of anti-patterns. Indeed, the refactoring of anti-patterns should be easier and less costly at modeling level than during implementation.

6.3 Threat to Validity

This section discusses the threats to validity of our study following common guidelines for empirical studies [94].

6.3.1 Construct Validity

In our anti-patterns study, we assumed implicitly that each anti-pattern is of equal importance, when in reality, this may not be the case. Future work must study the impact of the anti-patterns found in models in well-used dependent variables, such as class change- and fault-proneness, to assert whether all anti-patterns in models have a similar impact in the source code during implementation and maintenance.

6.3.2 Internal Validity

The UML Repository contains class diagrams collected from different categories. However, we still miss industrial models. Therefore, we ask companies to share their models for educational purpose.

The accuracy of Ptidej impacts our results. However, Ptidej has been successfully used in multiple studies [85][86][87][89], which have been ported to achieve high precision and recall [84]. However, other anti-pattern detection techniques and tools should be used to confirm our results.

In addition, the level of details of UML models affects the detection of anti-patterns in the class diagrams. It is possible that the lack of detailed information in class diagrams also affected the detection of anti-patterns. However, because the detection of these anti-patterns requires a high level of details in design (classes, methods, relations, and hierarchies (which are contained in the model)), we are confident about

the validity of our results. Yet, we will replicate our study with other techniques and tools in the future.

6.3.3 External Validity

We used two open source software projects in two studies, and made our conclusion based on these two systems. We use seven open source projects in another study and our conclusion is based on this data set. These systems that used in our studies are available in the UML Repository. It has different sizes and belong to different domains. Nevertheless, further validation on a larger set of systems is desirable, considering systems from different domains as well as systems from same domains.

6.4 Conclusion and Future Work

In this chapter, we describe a corpus study on the diagrams in the UML Repository. We show examples of an interesting relation between maximum coupling and size of the diagrams. More studies can be performed with this dataset, which can show behaviors of the designers, good patterns and bad patterns, also common patterns and anti-patterns.

We performed three experiments to investigate the relation between quality of the design and the source code.

In our study, we find that classes in the design have more changes and bugs than others. In the second, we investigated whether the design models produced by architects and designers before the implementation of software systems contain anti-patterns. We also examined whether the occurrences of the anti-patterns in models translate into the source code, affecting the same classes in models and the source code implementing these models. We conducted an empirical study on the prevalence of four anti-patterns: Complex Class, Large Class, Lazy Class, and Long Method, using both the architects' and designers' models of the seven systems (selected from the UML Repository) and the source code of these systems.

Our results showed that on average, 36% of the classes in the models that belong to anti-patterns also exist in the source code and also play roles in the same anti-patterns. Hence, we showed that anti-patterns appeared very early and concluded that architects and designers would benefit from help to identify and control these anti-patterns as early as possible.

Seven types of anti-patterns could be detected in the design phase: Complex, Large, Lazy, Blob, ClassDataShouldBePrivate, RefusedParentBequest, and BaseClassShouldBeAbstract classes. These anti-patterns mostly reappeared again in the source code in the same classes. Hence, it would be wise for maintenance teams to detect these anti-patterns early to save time and effort.

We found that classes in the design that have anti-patterns had more changes and bugs in the implementation. Therefore, anti-patterns should be detected and solved early in the design phase because in the source code it makes more changes and faults.

Future work includes analyzing more pairs of designers' models and their corresponding source code as well as analyzing more projects to propose prevention techniques. Because refactoring is easier at the design level, we aim to propose a technique to automatically refactor anti-patterns detected in models. For example, we envision that a Complex Class can be divided into two or more classes in models as well as in the source code. We will also consider anti-patterns as benchmarks for models quality and we plan to apply anti-patterns detection for whole models and systems in the UML Repository. We will make this information public to foster replications and contrasting studies.

Chapter 7

Quality Assessment of UML Class Diagrams

In this chapter, we present an experiment conducted for comparing how experts and students assess the quality of class diagrams. Six quality attributes were addressed: Understandability, Layout, Extensibility, Modifiability, Completeness and Correctness. From this study, we aim to find out how well students are capable of evaluating the quality of UML designs. One particular scenario that we have in mind that where students perform grading of peer-produced UML software design as part of a Software Engineering course. Moreover, we aim to learn which features experts and students use for assessing the quality attributes of class diagrams. The study reveals that experts and students' assessment of the six quality attributes differs significantly. However, a qualitative analysis of experts and students' feedback suggests that students use similar features as experts use for assessing the quality of diagrams. Hence, peer-feedback from students can be useful in educational settings.

This chapter is based on the following publications:

- Bilal Karasneh, Dave Stikkolorum, Enrique Larios, Michel R.V. Chaudron. **Quality Assessment of UML Class Diagrams - A Study Comparing Experts and Students-.** *MoDELS2015, Ottawa, Canada. 2015.*
- Bilal Karasneh, Michel R. V. Chaudron. **Online Img2UML Repository: An Online Repository for UML Models.** *In Proceedings of the 3rd International Workshop on Experiences and Empirical Studies in Software Modeling (EESSMOD@MODELS 2013), pages 61-66, Miami, USA. 2013.*

Quality is a multidimensional concept, and in practice people make different interpretations of the same concept. Nowadays UML [34] is the de-facto standard for modeling software systems. UML offers a rich set of symbols for describing software. Modern software designs contain many abstraction levels, and designing them is an iterative process [95]. The collection of design documents is an important part of the system documentation which will be used and maintained for a long time by a development organization. In the software engineering class, students should understand the importance of software models and their design process.

Software design is considered as a difficult task in comparison with programming for many students. One reason is that current Integrated Development Environments (IDEs) help students to improve the quality of their code, for example using code metrics such as a maintainability index and a cyclomatic complexity. On the other hand, current UML CASE tools do not give any hints to improve models, except some layout algorithms and syntax. Although there are some proposed tools that give students some feedback about their design, these tools still suffer from many limitations, such as availability and connectivity [96][97].

During programming courses, students are taught about the quality of the source code (including for example, naming and layout conventions and API design guidelines). In software engineering courses, students are taught to understand basic modeling concepts and modeling notations. For instance, what UML diagrams are, when to use a class diagram, how to create a sequence diagram, what are elements of use case diagrams. Many teachers focus on teaching students the proper use of syntactical elements in creating UML diagrams. In both programming and modeling, the completeness and correctness are key attributes of the quality of a solution. However, there are no specific rules or guidelines for assessing quality attributes of designs. This leaves students to self-learning on how to make a good design.

For proper learning of modeling and designing, students need to get feedback from their teachers, or from peers to evaluate their design. For example: this class should have more operations, this class name should be changed, this operation should have more parameters. One way of providing feedback could be to use a method for assessing the quality of UML models. Unfortunately, currently no such method exists.

In this study, we explore the use of ISO standards for software quality as a basis for reviewing UML models. Software product quality models such as ISO/IEC 25010 [11] have categories of quality characteristics, and each characteristic is composed of a set of sub-characteristics. One difficulty with this standard is that there are many ways of interpreting every characteristic.

In this chapter, we want to study the ability of students to evaluate their designs and other students designs. Also, we want to study whether the evaluations of students are consistent with those of experts. In addition to the quantitative analysis, we do a qualitative comparison of the feedback provided by students and experts.

To study this, we asked students to perform a modeling task, and then to evaluate their models and to evaluate models of other students in terms of six quality attributes:

Understandability, Layout, Extensibility, Modifiability, Completeness, and Correctness. Then we asked five experts to evaluate students models in terms of the same quality attributes.

In this chapter we address the following research questions:

- Can we trust students assessments and for the quality of UML class diagrams? Why?
- What kind of features that experts and students focus on when they measure the quality of UML class diagrams?

The aim of this study is to empirically investigate whether students evaluations are different from experts evaluation, and what are the differences and similarities between students' feedback and experts' feedback. The differences and similarities are useful to assess if the feedback of students can be useful for improving the quality of the design.

7.1 Related Work

We follow the general guidelines for experimental design and analysis from [98][99]. Tichy [100] shows that there are good reasons for conducting experiments with students, for testing experimental design and initial hypotheses, or for educational purposes. Depending on the actual experiment, students may also be representative of inexperienced professionals [101].

Boustedt [102] did an empirical study on how students understand class diagram using phenomenographic investigation. He found that the purpose of class diagrams and various elements of the UML notation were understood in a varied way. He recommended that teachers should put more effort in assessing skills in proper usage of the basic symbols and models, and students should have opportunities to practice collaborative design. Our experiment is different from [102] as we ask the students to evaluate class diagram directly in terms of six quality attributes, and we ask them to give feedback and explanation about their evaluation.

Ali et. al. [103] presented the UML class diagram assessor (UCDA) that evaluates class diagrams automatically based on their structure, correctness and language used. The aim of the proposed assessor is to guide students to represent class diagram correctly. The results of our experiment are useful for the kind of assessors in [103] because from the information collected we know which kind of feedback experts and students use for describing violations of modeling conventions and/or models improvement.

Hoggart et. al. [104] found that students understand UML design in classroom settings but find it hard to apply in exercises and tasks. They proposed a tool that gives students feedback about their diagrams in comparison with a model answer proposed by the student's teacher. Generating feedback based on model answers is a bit difficult

because in modeling there is typically more than one solution. Because it is difficult to find a sufficient number of experts, we decide to explore whether the feedback from peer students can help improve model quality.

Kaneda et. al. [105] show that class diagrams reflect the cognitive structure of English based cognitive linguistic. They found that there is impedance some mismatch for understanding of class diagram by students who are not native English speakers. In our experiment, our students are a mix of various nationalities. Before admission, our students have to pass an English (TOEFL) test, and therefore we consider that their English language skills will not influence our study too much.

Aguilera et. al. [106] show that names of elements in UML diagrams have a strong influence on their understandability. They proposed guidelines for naming various kinds of elements in UML.

Selic [107] shows that understandability is the most important characteristics of models. In our study, we show the features that experts and students focus on for assessing understandability of models.

7.2 Experiment Design

In this section, we explain our approach, the participants of the experiment and the evaluation form that the participants use for assessing class diagrams.

7.2.1 Approach

We conducted an experiment in Leiden University in which both experts and students participated. We gave the students a modeling task and asked them to use the StarUML CASE tool [25] to create their models. Upon completion of the modeling task, they had to upload their models to the UML Repository and evaluate their models based on six quality attributes. Also, they had to mention their background: academia, industry or both, and their experience in UML modeling (less than one year, <1-2>, <2-5> or expert). Subsequently, they had to evaluate other students' design based on the same six quality attributes. We asked students to explain their evaluations through feedback comments for each quality attribute. Students had a trial assignment two weeks before the experiment with another modeling task. This trial is important for the students to be prepared for the experiment. It helped them in getting acquainted with the type of assignment, the tools and thereby limits the learning effects. We also asked the experts to evaluate students' models based on the six quality attributes and to give a feedback of the models and describe their evaluations.

7.2.2 Participant

The participants are: five experts and 46 students.

7.2.2.1 Experts

Five experts joined this experiment. Each expert has at least five years of experience in UML modeling and software design. Two experts are teachers of software modeling and software engineering for at least three years. One of those two experts also worked in industry. The other three experts are PhD students in the area of software engineering since 2011.

7.2.2.2 Students

46 master students of the ICT in business M.Sc. program[1] in Leiden University participated in the experiment during their course on software engineering. All of them have less than one year experience in UML modeling. Some of them have some (mostly short) background in industry, but most of them have an academic background (just finished their B.Sc. degree).

7.2.3 Evaluation Form

The form for evaluating class diagrams was implemented through an online system. This system showed a form that contains:

1. The number of models that were evaluated by the participant (out of 46 models).
2. An image of a student's class diagram. The image is created by some other students and has not been evaluated by the participant before.
3. A list of radio-buttons for entering assessments for 6 quality attributes. Each quality attribute can be rated on a scale ranging from 1 to 8:
   - For Understandability, Extensibility and Modifiability: (1) is difficult, (8) is easy.
   - For Layout: (1) is complex, (8) is simple.
   - For Completeness: (1) is not complete, (8) is complete.
   - For Correctness: (1) is not correct, (8) is correct.
4. A comment box. For each quality attribute participants can submit details about their evaluation using a text box. We perform a qualitative analysis of the comments provided by experts and students to figure out which features they focus on when they assess the quality of a model.
5. A submit button. Stores the assessment and navigates participants to evaluate another design.

[1] This is a degree in the Science faculty of the University of Leiden. This degree is a mix of topics from Information System, Software Engineering and Management and Business Administration.

7.2.4 Modeling Assignment

The modeling assignment was about a library system. The modeling assignment, evaluation form, post-questionnaire and students designs are available in the supplemental materials of the experiments [108].

7.3 Comparing Model Evaluation

For comparing the evaluation between experts and students, we use a Multivariate General Linear Model (MGLM). This model is used because it considers multiple dependent variables and multiple independent variables. We also use bootstrapping [109], which is a method that approximates the sampling distribution of the sample mean. In our experiment, the dependent variables are the six quality attributes, and the independent variables are the assessors (experts and students scores). We use IBM SPSS [110] as statistics tool.

7.3.1 Experts Evaluation and Students *self* Evaluation

Each class diagram was evaluated by at least two experts and one student (each student evaluated his/her model). In the upload form, students were asked to evaluate their models. For making this comparison, we do resampling (bootstrapping) of 1000 times of size 121 for the experts evaluation and independently the same resampling time of size 46 for the students evaluation.

7.3.2 Experts Evaluation and Students *peer* Evaluation

From the set of evaluations, we leave out seven class diagrams because the variation (standard deviation) of student's evaluation is high. This leaves a total of 39 class diagrams. We have 95 model-evaluations from experts per each quality attribute. Moreover, each student evaluated at least class diagrams. For each quality attribute, we have 435 evaluations in total from students.

7.4 Results and Analysis

We did the quantitative analysis for experts and students evaluation, and qualitative analysis for their feedback and comments.

7.4.1 Quantitative Analysis

Figures 7.1 and 7.2 show the average evaluations of experts, and students' self-evaluation for understandability and layout respectively. The evaluation is sorted

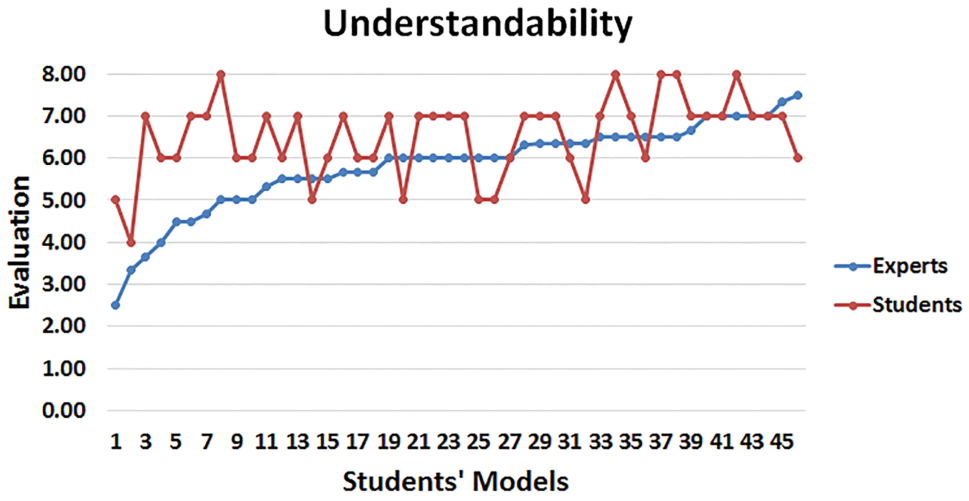

Figure 7.1: *Experts and students evaluation (self-evaluation) for Understandability*

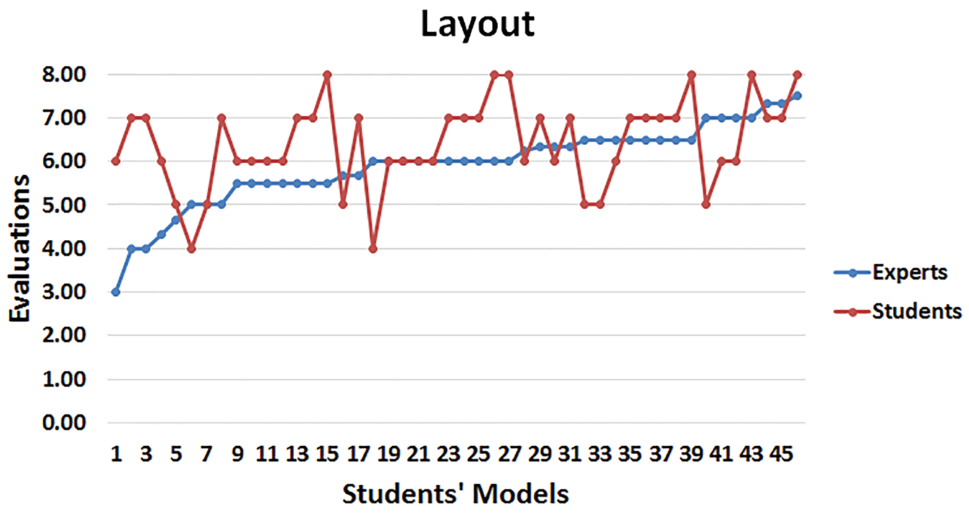

Figure 7.2: *Experts and students evaluation (self-evaluation) for Layout*

in ascending order based on experts' evaluation. Figures 7.1 and 7.2 show that experts and students differ in the most cases (high and low quality diagrams). Figures 7.3 and 7.4 show the average evaluations of experts and students' peer-evaluation for understandability and layout respectively. The evaluation is sorted in ascending order based on experts' evaluation. From Figures 7.3 and 7.4, students assessment is mostly

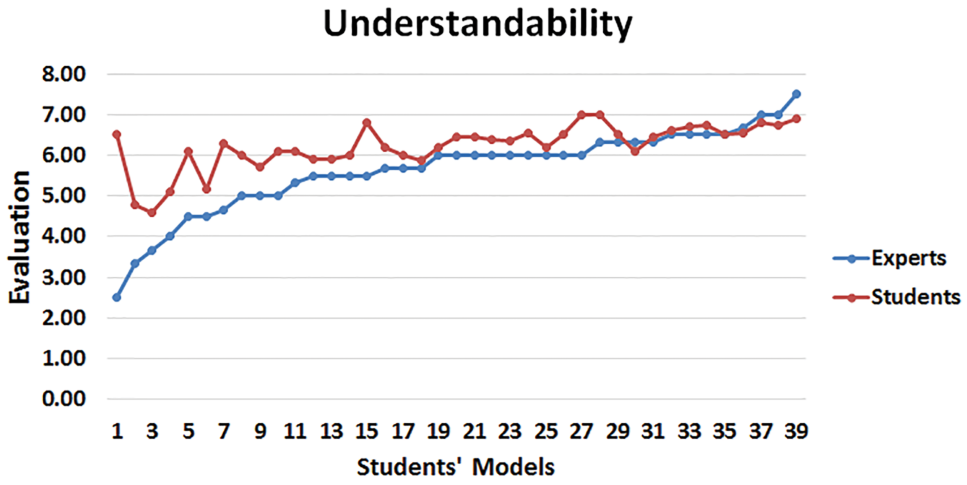

Figure 7.3: *Experts and students evaluation (peer-evaluation) for Understandability*

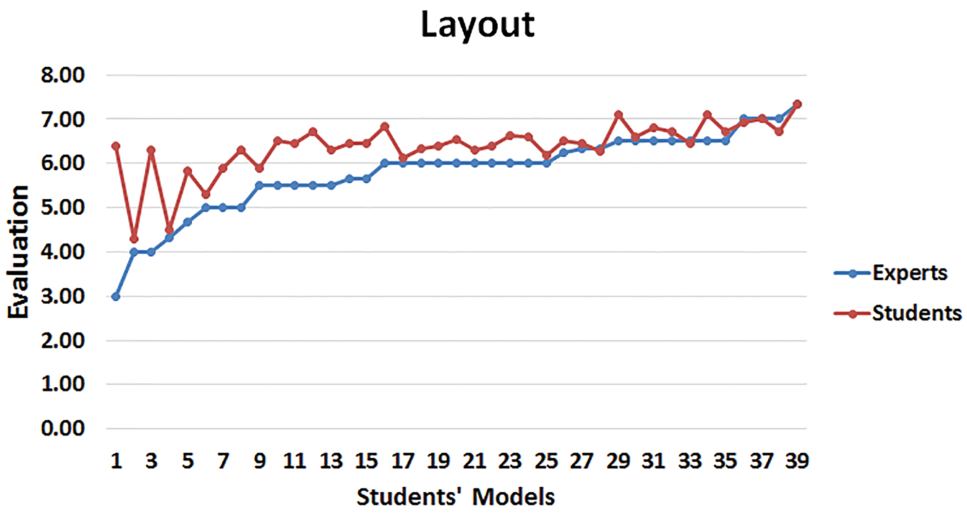

Figure 7.4: *Experts and students evaluation (peer-evaluation) for Layout*

higher than the experts' for understandability and layout, and sometimes they are close.

Table 7.1 shows the results of MGLM relating experts with students. Table 7.1 shows that there is a significant difference between expert evaluation and students self-evaluation. In addition, Table 7.1 also show that there is significant difference

Table 7.1: *Results of Multivariate General Linear Model*

Dependent Variable	Assessors		95% Sig.
Understandability	Experts	*Students Self-Evaluation*	0.00
		Students Peer-Evaluation	0.021
Layout	Experts	*Students Self-Evaluation*	0.003
		Students Peer-Evaluation	0.006
Extensibility	Experts	*Students Self-Evaluation*	0.003
		Students Peer-Evaluation	0.009
Modifiability	Experts	*Students Self-Evaluation*	0.001
		Students Peer-Evaluation	0.011
Completeness	Experts	*Students Self-Evaluation*	0.001
		Students Peer-Evaluation	0.053
Correctness	Experts	*Students Self-Evaluation*	0.00
		Students Peer-Evaluation	0.004

Table 7.2: *Description of Experts and Students Evaluation*

Dependent Variables	Experts	Self-Evaluation	Peer-Evaluation
	Mean	Mean	Mean
Understandability	5.71	6.51	6.22
Layout	5.74	6.4	6.38
Extensibility	5.59	6.18	6.14
Modifiability	5.52	6.2	6.06
Completeness	5.77	6.53	6.3
Correctness	5.07	6.31	5.86

between experts evaluations and students peer-evaluations. Table 7.2 shows the description of experts and students evaluations. From Table 7.2, all means of experts evaluations are less than the means of students evaluations in both cases (self/peer evaluation). For analyzing the evaluations, we show the correlation metrics between experts and students peer-evaluations in Table 7.3. In Table 7.3, it is possible to see many high correlations between quality attributes. First, the correlation between experts evaluation, understandability has a high correlation with all quality attributes. Second, on the student side, understandability also has a high correlation with most other quality attributes. We notice that the correlation between understandability and layout for students evaluations is higher than in experts evaluations. Third, regarding the correlation between experts' and students evaluations, the highest correlation is between experts understandability and students understandability. The second highest correlation is between experts and students evaluations for layout.

Table 7.3: *Correlation of Experts and Students peer-Evaluation*

	E_Unders.	E_Layout	E_Extens.	E_Modif.	E_Compl.	E_Correct.	S_Unders.	S_Layout	S_Extens.	S_Modif.	S_Compl.	S_Correct.
E_Unders.	1											
E_Layout	**0.74**	1										
E_Extens.	**0.77**	0.63	1									
E_Modif.	**0.84**	**0.71**	**0.88**	1								
E_Compl.	0.67	0.65	**0.70**	0.68	1							
E_Correct.	**0.71**	0.57	**0.83**	**0.85**	**0.74**	1						
S_Unders.	**0.70**	0.62	0.59	0.52	0.44	0.41	1					
S_Layout	0.57	0.67	0.56	0.53	0.47	0.47	**0.81**	1				
S_Extens.	0.61	0.57	0.62	0.53	0.47	0.45	**0.86**	**0.83**	1			
S_Modif.	0.60	0.62	0.57	0.56	0.51	0.37	**0.85**	**0.81**	**0.88**	1		
S_Compl.	0.54	0.37	0.49	0.46	0.58	0.51	0.62	0.56	0.59	0.56	1	
S_Correct.	0.59	0.45	0.58	0.56	0.66	0.61	0.62	0.59	0.60	0.61	**0.89**	1

7.4.2 Qualitative Analysis

We qualitatively analyze experts and students (peer-evaluation) comments/feedback for their evaluation. This analysis is important to see the features that experts and students use for assessing the quality of class diagrams. We discuss the feedback of three of the quality attributes: understandability, layout, and completeness. We choose these quality attributes because from Table 7.3, understandability (0.70) and layout (0.67) are the highest correlated quality attributes between experts and students peer-evaluation. In addition, understandability has the strongest correlation with others quality attributes. We choose completeness because from Table 7.1, experts and students almost differ with the significance of 0.053.

We use NVivo10[2] for qualitatively analyzing experts and students comments. In their comments, they explain how they evaluated models quality attributes, and features they used for their evaluation. Table 7.4 shows 11 features of models that experts and students used for assessing understandability. We observe that: (i) 64% of the features are used by both experts and students. (ii) 27% of the features are used by students but are not used by experts. (iii) 9% are used by experts and not used by students. Table 7.5 shows that experts and students used 12 features for assessing the layout, where: (i) 58% of the features are used by both experts and students. (ii) 9% of the features are used by students, but not used by experts. (iii) 33% used by experts, but not used by students. Table 7.6 shows that experts and students used 10 features for assessing completeness, where: (i) 60% of the features are used by both experts and

[2]http://www.qsrinternational.com/

Table 7.4: *Features that Experts and Students Focus on When They Evaluate Understandability*

Features	Experts	Students
Easy to read	-	X
Completeness	X	X
Extra information	X	X
Complexity	X	X
Correctness	X	X
Data type	-	X
implementation	X	-
Layout	X	X
Class, attributes and operation names	X	X
Relationship names	-	X
Number of classes, operations, and attributes	X	X

Table 7.5: *Features that experts and students focus on when they evaluate Layout*

Features	Experts	Students
Classes Hierarchy, alignment	X	X
Classes with similar size	X	-
Complexity	X	X
Number of classes, attributes and operations	-	X
Distance between Classes	X	X
Rectilinear edges and diagonal edges	X	-
Line Style (overlapping, crossing, bend)	X	X
Good Class name	X	-
Neat or chaotic structure	X	X
Easy to read	X	X
Same Layout for same/All relationships	X	-
Extra information	X	X

students. (ii) 30% of the features are used only by students. (iii) 10% are used only by experts. Although we see experts focus on more features in Table 7.5, it may be that students are interested in many of the same features – yet they do not mention them clearly in their feedback.

Table 7.6: *Features that experts and students focus on when they evaluate Completeness*

Features	Experts	Students
Model Abstraction	-	X
Functionality	X	X
Strange Relationships	X	X
Missing Classes, Attributes, and operations	X	X
Data types	-	X
Multiplicity	X	X
Functions Parameters	-	X
Relationship names	X	X
Requirements	X	X
Model Semantics	X	-

7.5 Discussion

From the MGLM results in Table 7.1, we conclude that there is a significant difference between the evaluation of experts and students. The results also show that peer-evaluation of students is closer to the evaluation of experts than self-evaluation (because the mean difference is bigger between experts and self-evaluation than with peer-evaluation for all quality attributes as shown in Table 7.2). We explain this by the different viewpoints in the peer-evaluation. Different points of view may have caused different evaluations that on average became more reliable, or at least better than the self-evaluation.

The qualitative analysis of experts' and students' comments shows that students use most features that experts use for assessing the quality of class diagrams. In Table 7.4, 7.5 and 7.6 we summarize the features that experts and students use for assessing understandability, layout and completeness respectively. In the qualitative analysis, we only take into account the issues that can be clearly identified in the feedback. We notice that feedback from experts is more specific than that from students. For example, some students mentioned they did not like a class, but they did not mention what was the problem with this class: name, size, position, etc. However, this general feedback can still be useful because it can be considered as general hints that direct students to a particular area where they still themselves need to find out what needs to be improved.

We conclude that students largely use similar features for assessing the quality of class diagram as experts use. Hence, their feedback is useful for improving their models. So we expect that if students exchange their feedback about their models, this will be a valuable source of feedback for learning and improving their models. Also, we expect that students can make a better evaluation if they do this in a group because they can then discuss their different viewpoints and improve their evaluation.

In addition, students (peer-evaluation) is not so close to expert evaluation. Students seem reluctant to fail fellow students.

7.6 Threats to Validity

In this section, we discuss the threats to validity of our study.

7.6.1 Internal Validity

We ensured that students are familiar with class diagrams. The experiment was conducted at the end of the Software Engineering course where they had a trial two weeks before the experiment. The participants did not know the aim of our experiments, nor the measures that we are looking for, in order to avoid their expectations from biasing the results.

7.6.2 External Validity

There were 46 students participants in the experiment. To mitigate their representativeness, we only address their experience level with UML, and their background (academic, industry or both). About the modeling task, we chose a system from an application domain that should be familiar to students, which is library system.

7.7 Conclusion and Future Work

In this chapter, we presented an experiment that investigates the difference between experts and students in assessing the quality of UML class diagram empirically. We made two comparisons: first between experts and students' self-assessment. Second, between experts and students peer- assessment. We use the Multivariate General Linear Model as a statistical method for making those comparisons. The results show that experts and students (self- assessment) are different in terms of means (95% significance). The students self- assessments are higher than experts assessments in terms of mean for the quality attributes used in the experiment. The results also show that experts and students (peer-evaluation) are different in terms of mean (95% significance). The students peer- assessments are higher than experts assessments in terms of mean for all quality attributes used in the experiment. Analyzing the correlation between experts' assessments and students peer- assessments shows that understandability is the highest correlated quality attribute, and that layout is the second highest. The correlation also shows that understandability is correlated with most of the other quality attributes based on both experts assessments and students assessments.

We did a qualitative analysis of experts' feedback and students feedback in peer-assessments. From this, we observe that students mostly use similar features as experts for their assessments. So we conclude that feedback from students is valuable and can be useful for other students for improving their designs.

In the future, we are planning to replicate the experiment and ask students to assess the quality of class diagram in groups. We believe that having an online community for students where they can exchange their models, and their feedback is very useful for improving modeling education. So we are establishing this community with the collaboration of some experts. From this community, students and experts can upload their models and exchange their feedback.

Chapter 8

Using Examples for Teaching Software Design

In this chapter, we present a research that is positioned in the field of software design method and teaching thereof. The aim of this research is to study the effects of using a collection of examples for creating a software design. We ran a controlled experiment for evaluating the use of a broad collection of examples for creating software designs by software engineering students. In this study, we focused on software designs as represented through UML class diagrams. The treatment is the use of the collection of examples. These examples are offered via a searchable repository. The outcome variable we study is the quality of the design (as assessed by a group of experts). After this, all students were offered the opportunity to improve their design using the collection of examples. We ran a post-assignment questionnaire to collect qualitative data about the experience of the participants. Considering six quality attributes measured by experts, our results show that: 1) the models of the students who used examples are 18% better than those of who did not use examples. 2) the models of the students who did not use examples for constructing became 19% better after updating their models using examples. We complement our statistical analysis with insights from the post assignment questionnaire. Also, we observed that students are more confident about their design when they use examples. Students deliver better software designs when they use a collection of example software designs.

This chapter is based on the following publications:

- Bilal Karasneh, Rodi Jolak and Michel R. V. Chaudron. **Using Examples for Teaching Software Design.** *In Proceedings of the 22st Asia-Pacific Software Engineering Conference (APSEC2015). New Delhi, India. 2015*
- Bilal Karasneh, Michel R. V. Chaudron. **Online Img2UML Repository: An Online Repository for UML Models.** *In Proceedings of the 3rd International Workshop on Experiences and Empirical Studies in Software Modeling (EESS-MOD@MODELS 2013), pages 61-66, Miami, USA. 2013.*

Examples could guide students who are novices in UML modeling to create better designs. In fact, modeling is much more difficult for students compared to programming tasks. This difficulty occurs because for most programming languages, students can get feedback immediately from the harsh compiler on the code they produce, e.g. through compile and run-time errors. However, such feedback is poorly available in modeling, and current CASE tools do not give hints for users related to models quality. Therefore, students need to have some feedback from their teacher, or at least from each other to trust their designs. As a consequence, students cannot get the kind of self-learning facility when they study modeling as they can get for learning programming.

We see that a variety of examples could improve students' performance for creating designs. From another side, a variety of examples could make students confused and make the modeling task more difficult to perform. We conducted a controlled experiment and asked students to give their feedbacks about using examples for creating software designs.

We offered students our UML Repository [111][112], to find examples of class diagrams. We asked students to compare between using the model repository, and searching for useful examples on the internet. We also asked the students how they preferred to search for example class diagrams.

The results show that using examples helps most of the students for creating and improving their designs, and it makes them more confident. A few students felt confused when they used examples because they could not distinguish between good and bad examples. The results also show that most of the students prefer using the UML Repository more than searching the internet to find models. This is because the repository produces more specific results because it supports searching based on details of the contents of UML models, e.g. names of classes and operations. It is difficult to search for these features using generic internet search engines. As a result, this specialized search saves students' time and effort.

8.1 Related Work

Van Gog et al. [113] showed that using examples decreases the mental effort required to understand problems. On the other hand, Goldstone and Son [114] showed that examples capture intuition. Seater [115] stated that although examples may include incorrect solutions beside the correct ones, understanding them gives additional insight and helps to reason why and when the model is correct. Many studies come up with the fact that solving problems is more effective when having multiple examples [114][115][116].

Bkak et al. [117] presented the Example-Driven Modeling (EDM) approach, which uses explicit examples for creating and validating business knowledge. They present many questions related to EDM, like: How useful are the examples for building models? What is the impact of examples on the comprehension of models? What kind of tool support is needed to work with examples?

Many empirical studies were conducted to evaluate comprehension techniques in UML. Zayan et al. [118] constructed a controlled experiment for empirical evaluation of EMD. They represented the abstraction part as UML class diagram and the examples as object diagrams. The results showed that EDM is better than having model abstraction only. Many studies take the impacts of the layout on model comprehension into account, and most of them use controlled experiments with different methods, like eye tracking, questionnaires and surveys [119]. Störrle [120][121] studied impacts of models layout from different perspectives. Nugroho [122] illustrated the effectiveness of the levels of detail on the UML model comprehension.

Basing on models comprehension, and by this experiment, we want to see whether multiple examples having different layouts and different levels of detail can help students to create better models and use elements of UML in the right way. In addition, we want to know if they could learn from examples how many details to include in their designs, make them more confident and optimize their model.

Our UML Repository [111][112] contains more than 810 UML class diagrams collected from the internet, open source projects, collaborations and via scientific literature. There are no other published UML models yet. Models are available as images and XMI. The available class diagrams vary in size and complexity. This repository is built using our Img2UML tool [52][53] that converts UML diagrams stored in image formats into XMI.

In our experiment, we use the UML Repository where models are searchable. Users can search for models using class-, attribute-, and operation names. In addition, users can search for models based on design metrics, for example: the number of classes.

8.2 Research Questions and Hypotheses

In this section, we explain our research questions and hypotheses that we are going to test. We address the following research questions:

- RQ1: Does the aid of using examples improve the quality of software design models created by novices?
- RQ2: Does using a variety of examples affect the quality of models created by novices?
- RQ3: What is the best way to offer examples? Can a model-repository be considered a good way of offering model examples?

Accordingly, we formulate the following *null*/**alternative** hypotheses:

- $H1_0$/**H1**: Constructing models with the aid of examples [*does not improve*]$_0$ /[**improves**] the quality of models created by novices.
- $H2_0$/**H2**: Improving models with the aid of examples [*does not improve*]$_0$ /[**improves**] the quality of models created by novices.
- $H3_0$/**H3**: Using a variety of examples [*does not improve*]$_0$ /[**improves**] the quality of models created by novices.
- $H4_0$/**H4**: Using a variety of examples [*does not improve*]$_0$ /[**improves**] the quality of models improved by novices.
- $H5_0$/**H5**: Constructing models with the aid of examples [*does not improve*]$_0$ /[**improves**] the students perceived confidence in their created models.
- $H6_0$/**H6**: Using model repositories can be considered as [*normal*]$_0$ /[**better**] way of offering model examples.

In the hypotheses, we differentiate between constructing/creating and improving models with aid of examples. Improved models are models updated by students later. We consider that the students who do not improve their models are confident about these models, which they created during the experiment. Quantitative and qualitative analyzes were performed to accept or reject the hypotheses.

8.3 Experiment Design

The aim of the study is to investigate the effect of using model examples in creating designs. We followed the guidelines for software engineering experimentation [123]. Figure 8.1 shows the experimental approach.

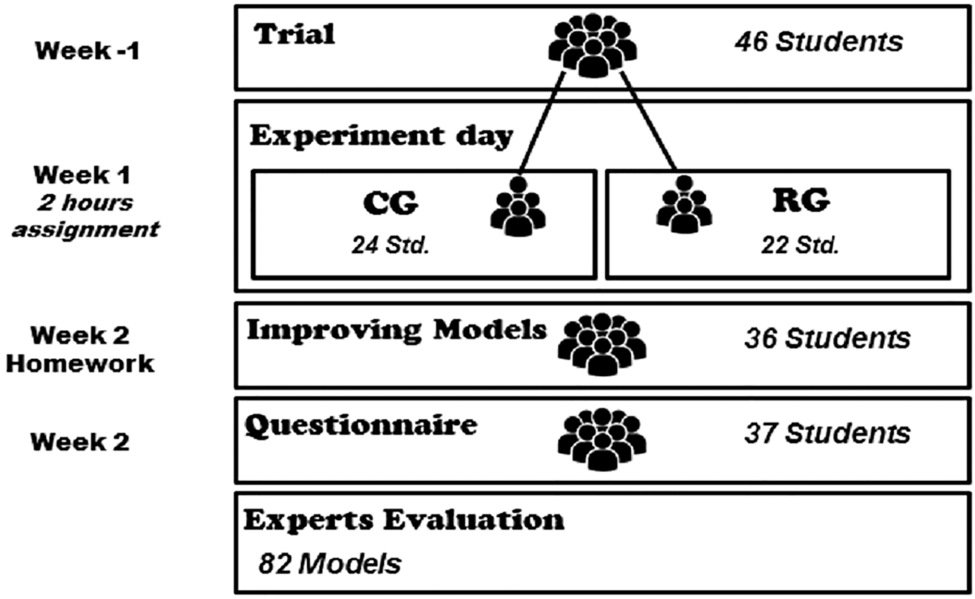

Figure 8.1: *Experimental Approach*

8.3.1 Method

The experiment was conducted at Leiden University; both experts and students participated. First, students were given a modeling task, and they used the StarUML CASE tool [58], to create their models. The students were separated into two groups: one group was allowed to search for models available in the UML Repository, and we call this group the Repository Group (RG). The other group was not allowed to use any examples, and we call it the Control Group (CG). Before starting with the experiment, the students did not know in which group they were. Students also had a two weeks trial before the experiment with another modeling task. The trail was important to get them familiar with using the repository. Students were given two hours to create their models. After that, we gave all students from both groups two days to improve their models using the repository. For CG, they had the opportunity to use the repository to improve their models. For RG, this simply meant that they had more time to use the repository to improve their models. After that, we asked the students to answer an evaluation questionnaire.

8.3.2 Operation

The entire experiment including the trail, constructing models, improvements, the questionnaire and experts' evaluation covered a period from the middle of November

2014 until the middle of February 2015. The experiment had been scheduled at the end of a software engineering course.

8.3.3 Evaluation

Five experts from Leiden and Chalmers Universities joined this experiment; they are working in academia and have some experience in the industrial domain. Two of them are also expert in teaching courses like software engineering and software modeling. We asked them to evaluate the students' models basing on six quality attributes (scale from 1 8): understandability, layout, extensibility, modifiability, completeness and correctness. Experts evaluated all of the 82 models individually. Experts did not know whether a certain model belongs to the RG or CG group. Moreover, they did not know which models were created or improved during which phase of the experiment.

8.3.4 Participant

Forty-six students from the Master program "ICT in Business" from Leiden University participated in the experiment. All the students, at that time, had less than one year of experience in UML modeling. The students were randomly divided into two groups, 22 students were assigned to the RG, and 24 students to the CG.

8.3.5 Data Collection

Students models, questionnaire answers, and models evaluation were collected. All of these data were available at the repository. First, the students uploaded their models to the repository, and they answered two questions: 1) about their background (academia, industry or both), 2) about their experience in UML modeling whether it is less than one year, <1-2>, <2-5> or more than 5 years. Second, students could update their models and upload them again. Third, the students answered a post-experiment questionnaire. The questionnaire contained 12 questions: seven questions with a scale (-4,4) or (1,8), four open questions, and one multiple choice. Finally, the experts used an online evaluation form. The evaluation form shows a class diagram created by a student and six quality attributes, each one of them has a scale from 1 8:

- For Understandability, Extensibility and Modifiability: (1) is difficult, (8) is easy.
- For Layout: (1) is complex, (8) is simple.
- For Completeness: (1) is not complete, (8) is complete.
- For Correctness: (1) is not correct, (8) is correct.
- Uploading, evaluation, and questionnaire forms are available in the supplemental materials of the experiment [108].

8.4 Results

In this section, we describe the results of a quantitative analysis based on the experts' evaluation. We also describe the outcomes of qualitative analysis based on the post-experiment questionnaire.

To check the normality of both populations (RG and CG), we took the average of experts' evaluation per each quality attribute, and then we used the Shapiro-Walk test [124]. When both populations are verified to be normally distributed, the independent samples students t-test [123] was used to check whether there is a significant difference between the mean of both populations. However, when the populations are not normally distributed; the non-parametric Mann-Whitney test [125] was used for the same comparison of the mean of both populations.

We used the statistical package R [126] to perform all tests. We chose a significance level of 0.05, which corresponds to a 95% confidence interval.

8.4.1 Experts Evaluation

In this section, we show the results of the experts' evaluation for both RG and CG, and the experts evaluation for CG before and after they use the repository.

8.4.1.1 Comparison between the RG and CG

Normality of all evaluations was checked, and the assumption was met for all cases. RG models have in average an 18% better evaluation of all quality attributes than CG models. Table 8.1 shows the one-tail t-test results for all quality attributes evaluations. All p-values in Table 8.1 show that there is a significant difference between RG and CG, and the RG has better results. Figure 8.1 shows the distribution of scores of RG and CG as boxplots.

8.4.1.2 Analyzing the improvement by the control group

Only 20 students from 24 improved their models in the CG, so we made the comparison based on the experts' evaluation for the models created by those 20 students. The normality of all evaluations was checked, and the assumption was met for understandability, modifiability, and correctness. After updating their models with the aid of examples, the CG received 19% better evaluation of all quality attributes compared to what they got for their old models created before using examples. Table 8.2 shows the results of the one tail test for all quality attributes. From Table 8.2, we observe that all p-values, except for the correctness case, indicate a significant difference for CG before and after using the repository, and there are better results after using the repository. For correctness, there are better results after they used the repository in

Table 8.1: *Results of Students t-test one tail*

Quality attribute	Groups	Mean	Median	p-value
Understandability	RG	5.56	5.58	0.001
	CG	4.61	4.83	
Layout	RG	6.12	6.25	0.003
	CG	5.09	5.42	
Extensibility	RG	5.64	5.67	0.009
	CG	4.66	4.83	
Modifiability	RG	5.53	5.58	0.00
	CG	4.52	4.67	
Completeness	RG	5.77	6.13	0.006
	CG	4.76	5	
Correctness	RG	5.25	5.42	0.024
	CG	4.46	4.67	

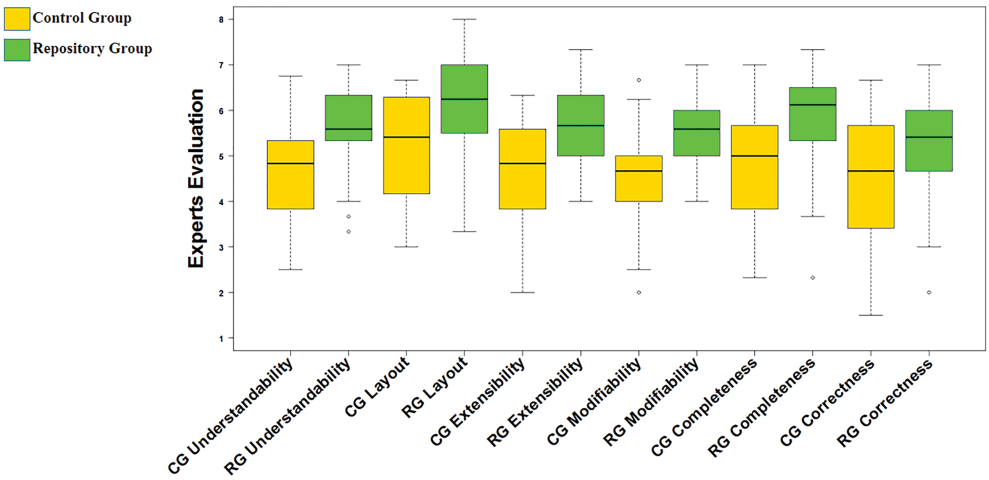

Figure 8.2: *Evaluation of RG models and CG models*

terms of the median. Figure 8.3 shows the evaluation of CG before and after they used the repository.

8.4.1.3 Comparing RG and CG after the CG Improves Their Models

We could not find any significant difference between evaluations of RG models created during the experiment, and improved models of the CG. It makes us conclude that

Table 8.2: *Results of students t-test and Mann Whitney test (One tail) of the CG*

Quality attribute	Groups	Mean	Median	p-value
Understandability	After	6.03	6.33	0.000
	Before	4.71	4.83	
Layout	After	6.05	6	0.024
	Before	5.25	5.58	
Extensibility	After	5.89	6	0.000
	Before	4.69	5.13	
Modifiability	After	5.78	5.83	0.000
	Before	4.62	4.67	
Completeness	After	5.94	6.17	0.000
	Before	4.83	5	
Correctness	After	5.18	5.33	0.068
	Before	4.63	4.67	

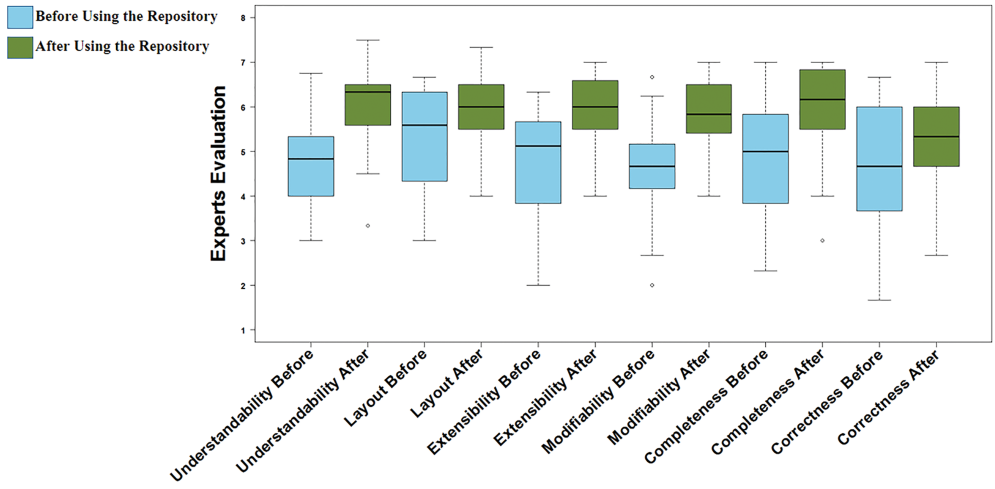

Figure 8.3: *Evaluation of CG models created during the experiment and after their improvement using the repository*

there is no difference of using examples during the construction of models or during the improvement.

8.4.2 Models Improvements

We instructed the students not to improve/update their models when they already think that they have good design. We assume that the students who did not improve their models were confident about their models. From the 46 students that participated in the experiment, only 36 students improved their models: 16 from RG and 20 from CG. Quantitatively: 27% of the students from RG were confident about their models, and 16% of the students from CG were confident about their models.

8.4.3 Post-assignment Questionnaire

In this section, we discuss the highlights of the responses of the students to the post-assignment questionnaire. 37 students responded to the questionnaire, 17 are from RG, and 20 are from CG. Next, we discuss the questions and responses of the post-assignment questionnaire:

1. *Do you think that using examples of class diagrams helps you to make a better design?*
   Type of the Answer: A Likert scale from -4 = not help to 4 = helps a lot without a (0) option.
   Analysis: Figure 8.4 shows the histogram of students' answer to this question. 86% of the students stated that using examples helps them in creating their design.

2. *Do you find it useful that the repository contains multiple examples for the same application/domain?*
   Type of the Answer: A Likert scale from 1 = not useful to 8 = very useful.
   Analysis: Figure 8.5 shows a graph of the student responses: 89% of the students stated that having multiple examples is very useful.

3. *How do you rate the relevance of class diagrams you found in the repository to your own design assignment?*
   Type of the Answer: A Likert scale from -4 = not relevant to 4 = very relevant without a (0) option.
   Analysis: Figure 8.6 shows that 94% of the students found relevant class diagrams in the repository related to their assignment.

4. *How do you rate the quality of class diagrams you found in the repository?*
   Type of the Answer: A Likert scale from -4 = poor quality to 4 = good quality without a (0) option.
   Analysis: Figure 8.7 shows that 86% of the students said that models in the repository have good quality. From their comments for this question, some students measured the quality based on the information within the assignments. Some students found some mistakes in some diagrams, but most of the models that they had explored have good quality.

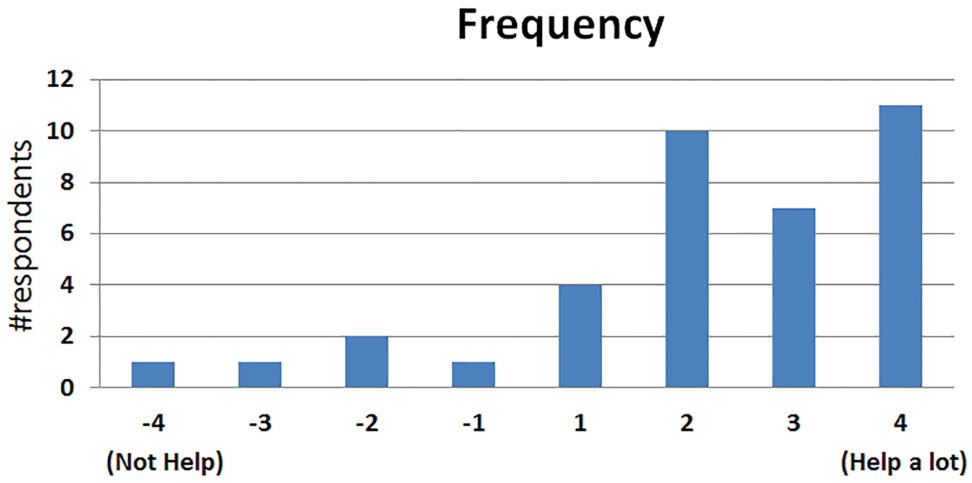

Figure 8.4: *Using examples of class diagrams helps to create better design*

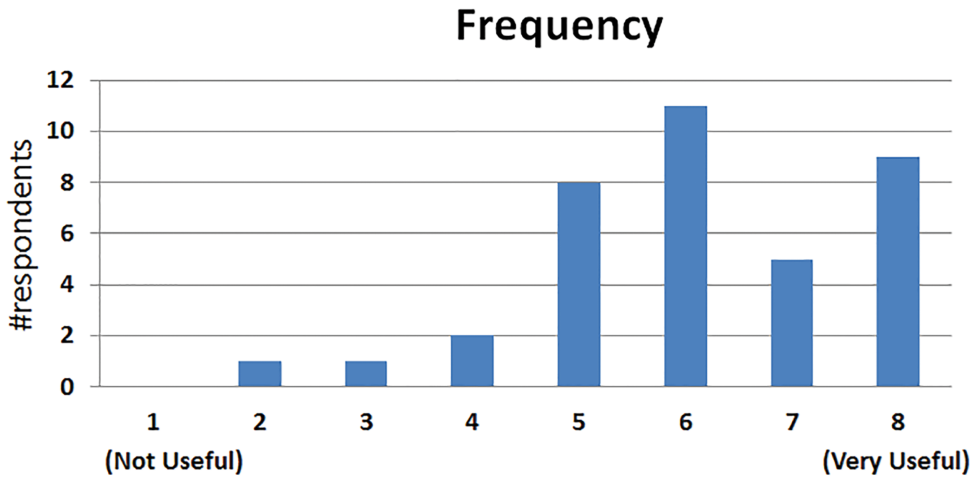

Figure 8.5: *Usefulness of having multiple examples for the same application domain in the repository*

5. *Do you think using the UML repository is more helpful to you than searching for examples of UML diagrams on the internet?*
   Type of the Answer: A Likert scale from -4 = not help to 4 = help a lot without a 0 option.
   Analysis: Figure 8.8 shows that 92% of the students said that using the model repository is more helpful than searching for models on the internet.

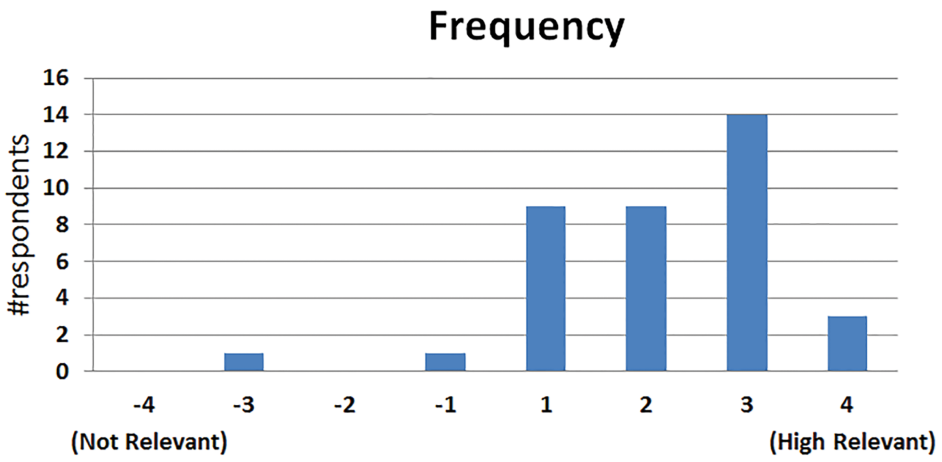

Figure 8.6: *Rate of the relevant class diagrams found in the repository to the design assignment*

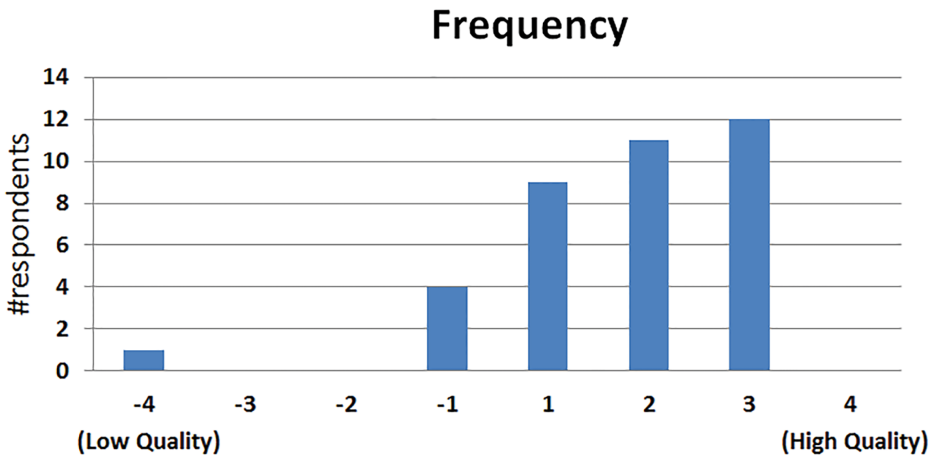

Figure 8.7: *Rate of the quality of class diagrams found in the repository*

6. *How do you rate the usefulness of searching based on class-, attribute- and operation-names to find relevant class diagrams?*
   Type of the Answer: A Likert scale from 1 = not useful to 8 = very useful.
   Analysis: Figure 8.9 shows that 95% of the students said that searching based on class-, attribute- and operation names is very useful.

7. *Did you find the available search techniques efficient to help you to find class diagrams?*
   Type of the Answer: A Likert scale from -4 = not efficient to 4 = very efficient without a (0) option.

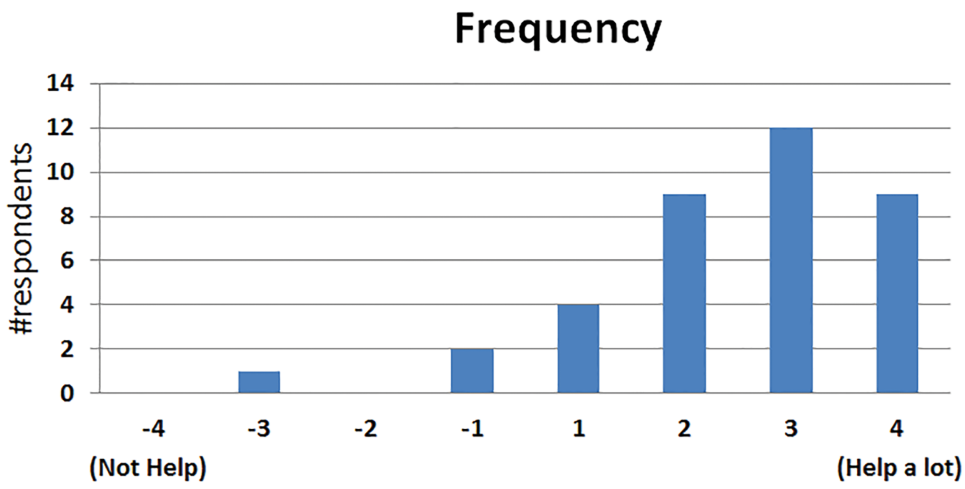

Figure 8.8: *Using UML Repository is more helpful than searching for examples on the internet*

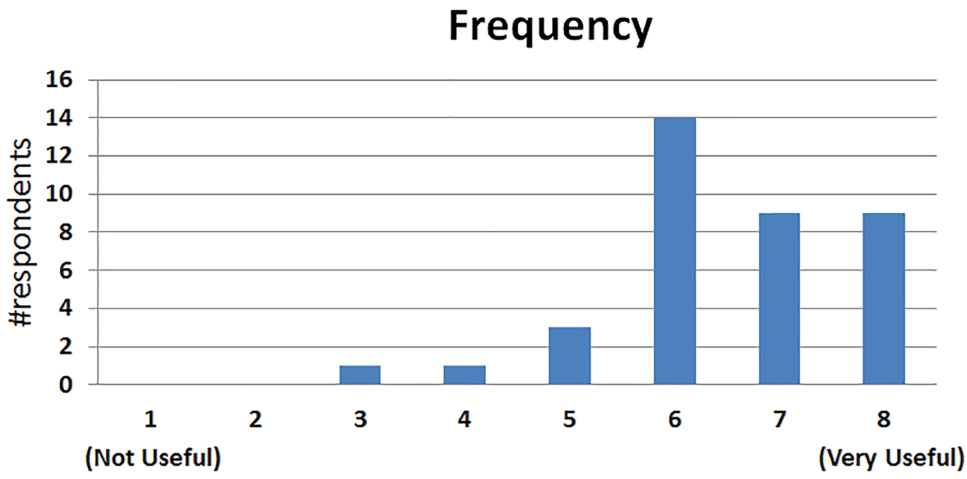

Figure 8.9: *Usefulness of searching based on class-, attribute- and operation names for finding relevant models*

Analysis: 89% of the students said that the available searching techniques in the repository are efficient.

8. *Are there another types of searching for class diagrams that you would like to see?*
   Type of the Answer: Open question.
   Analysis: 62% of the students stated that there is no need for other type of search-

ing mechanism. 22% of the students prefer to add a category-based searching mechanism. 14% prefer to add a search mechanism based on the description of class diagrams. 2% of the students asked to add a search mechanism based on relationship labels.

9. *Which aspects of the repository did you find easy to use in creating your own class diagram?*
   Type of the Answer: Open question.
   Analysis: 76% of the students stated that the search mechanism, in general, is the easiest aspect to use in the repository. 15% stated that multiple examples are the aspect that is the easiest to use. 9% stated that the search mechanism using class names is the easiest aspect to use.

10. *Which aspects of the repository did you find useful in creating your own class diagram?*
    Type of the Answer: Open question.
    Analysis: 49% of the students stated that searching is the most useful aspect in the repository. 44% of them considered the examples in the repository are the most useful aspect. 7% of students said that the easiness in using the repository is the most useful aspect.

11. *Which information from the class diagrams that you found in the repository did you use for making your own design?*
    Type of the Answer: Open question.
    Analysis: 64% of the students stated that relationships are the information they found and used for the design. 43% stated naming as the most useful information they found and used. 40% of the students stated that the syntax is the most useful information they found and used. Finally, 21% of the students said that roles and the structures of class diagrams are the most useful information they found and used.

12. *During the experiments, how much time did you spend using the repository?*
    Type of the Answer: Multiple choice, (<10 Min), (<10,20>), (<20,30>) and (>40).
    Analysis: The answer of this question is regarded to models improvement/updating, where both RG and CG use the repository. Figure 8.10 shows that 24% of the students said that they used the repository less than 10 minutes. 32% of the students said that they used the repository from 10 to 20 minutes. 30% of the students said that they used the repository from 20 to 30 minutes. 14% used the repository for more than 40 minutes.
    Figure 8.11 shows that 76% of students of RG spent less than 40 minutes using the repository. On the other hand, Figure 8.12 shows that 95% of students of CG spent less than 40, and 30% of the students used the repository for less than 10 minutes. Therefore, we conclude that students spent less time using examples when they use it only for updating their models in comparison with using it for creating and updating.

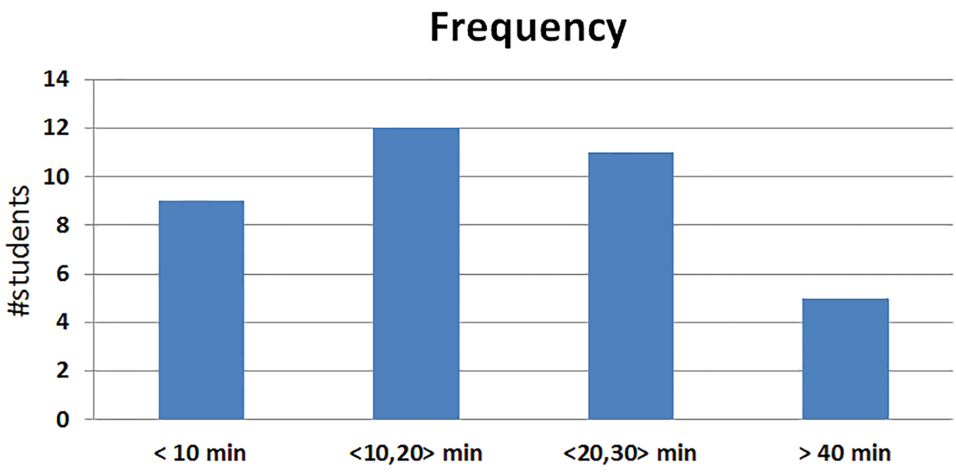

Figure 8.10: *Time spent by students using the repository*

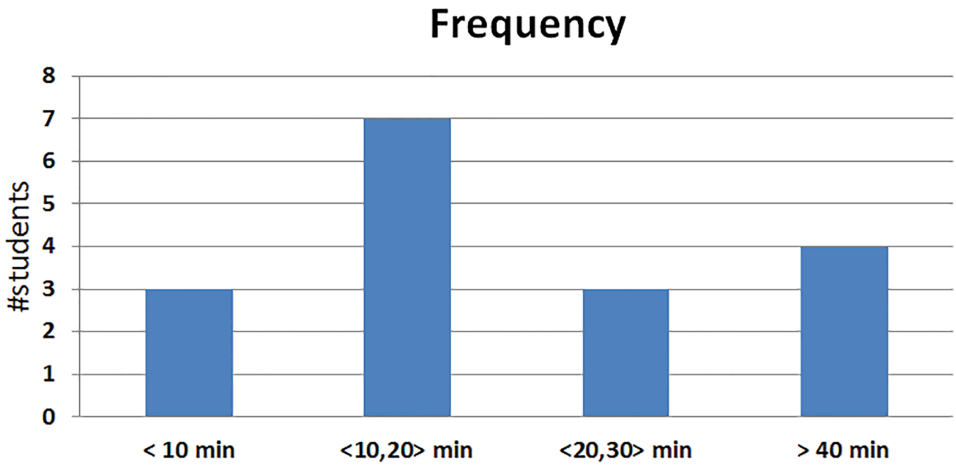

Figure 8.11: *Time spent by students using the repository*

8.5 Discussion

In this section, we discuss the results based on the aforementioned research questions, and we explain the results with more details considering the answers of the students to the questionnaire.

Table 8.1 shows that students from RG achieved higher evaluations than CG. This guides us to accept that constructing models with the aid of examples improve the

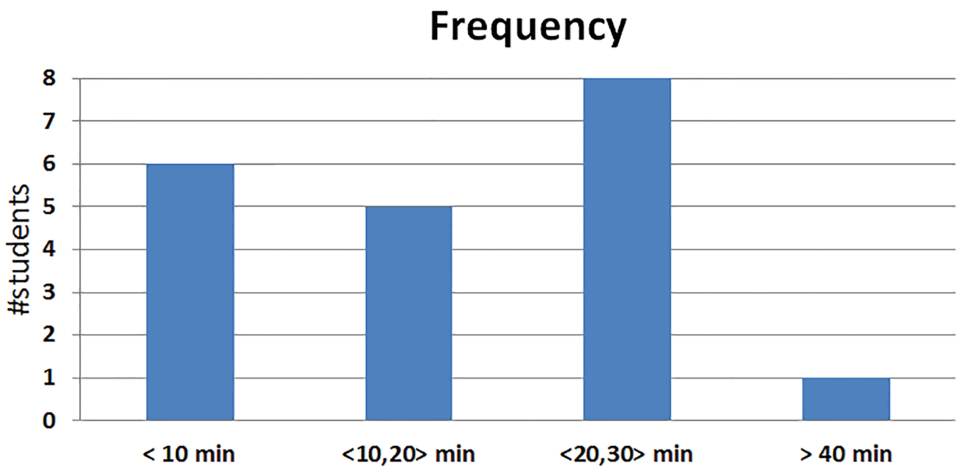

Figure 8.12: *Time spent by students using the repository*

quality of models created by novices compared with others who do not use examples. From Table 8.2, students of the CG achieved higher evaluation for their improved models using the repository compared with their old models. This also shows that using examples helps novices to improve qualities of their models.

Regarding the repository questionnaire, most of the students stated that using examples is really helpful. From their comments, we summarized that the repository of examples helps in the following manners:

- It gives the students a direction or starting point for creating their models.
  "Yes, even with some errors in the models (of course I didn't expect we could find the answer in the repository) it gave me a direction"

- It gives students many ideas from a different perspective as there is no optimal solution.
  "Yes, by seeing other examples and others models. And maybe see it from a new or different perspective which improves your diagram"

- It helps students to correct mistakes in their design.
  Especially when at a beginner level, using the repository helps with the design, ideas that I may have missed like level of abstraction, or solving a certain problem. Also it helps to spot mistakes and minor flaws"

- It helps them to complete their diagrams and reminds them regarding some details.
  It helps me to make a more complete diagram. It reminds me of some details"

- It helps in creating models more easily and more quickly.
  You can make your model more easily and quicker with the help of the repository"

A few students disagreed that using examples is helpful. We summarize their comments into two main categories:

- Constructing models with the aid of examples in general makes them lazy, but they agree that using examples improves their models.
  I think that using examples does not really help. It makes you lazy and does not force you to really learn about UML and Software Engineering"

One possible lesson could be to choose the assignment such that there are no models in the repository that are close to the same application domain nor close to the solution. If the repository grows, it may be more practical to restrict the diagrams that students get access to; hence to exclude some diagrams from being found.

- For some students, using examples confuses them. They prefer working without examples because they cannot distinguish whether the examples are good for their task or if they are of good quality or not.
  "It helped me a bit. I don't have a lot of experience with UML and this helped me to see and use it as a reference. It made me also doubt myself, because they were all so different"

Most of the students prefer to see multiple examples, because:

- They can pick out elements from different models, and make a new one.
  "Yes I did, in that way you have multiple perspectives. You can pick out the best elements from each model and make a personal one with this information and my basic knowledge (there is not one solution)"

- Compare models and align:
  "You can see what is usual for this application/domain to align your own model with a kind of standard"

- Multiple examples give multiple depictions of a certain subject, and show important functionalities which are not taken into account.
  "A variety of examples regarding a specific topic is always helpful, as the user has access to multiple depictions and implementations of a certain subject and he may trace functionalities that he might not have foreseen as necessary"

A few students do not like to see multiple examples, because it makes them more confused:
"It is useful to see multiple examples for the same domain but it also creates some doubt because you don't know if your modeling way is the right way"

For rating the relevance of class diagrams in the repository to the design assignment, students stated that they can find relevant examples to their assignments. Students

stated that it is enough to give a direction and to create their model.
"Helpful enough to give directions to use for our design"

One student commented that he dislikes using examples because it makes students lazy as they make use of examples to create their models.
"Everyone designs software according to his own view. Therefor I think you should try to model as much as possible by yourself. I also consider taking someone else's UML design as lazy"

For rating the quality of class diagrams in the repository, students stated that models that they found in the repository are helpful to give them directions for their design. Some examples are not correct and have some mistakes. Nevertheless, students learned how to skip or correct these mistakes. A few students stated that it is not easy to find good examples. We infer this from the keywords they used for searching, some keywords help to retrieve good relevant models.
"Most of the diagrams have a clear and understandable structure, which makes it easy for the user to extract any information that he wants"
"Some were of really high quality but there were also some models that contained some errors"
"A lot of diagrams made no sense, were incomplete or were not usable"

In response to the searching in the UML Repository is more helpful that searching on the internet, students stated that they prefer using the UML repository. Their preference is motivated as follows: Multiple examples are easily accessed, and the possibility to direct the searching mechanism to target class-, attribute- and operation-names gives advantages to the repository.
"Having the ability to search by keywords in the UML repository is as effective as using a search engine, plus it has the advantage that it has a variety of examples implemented in different ways and styles"

For the usefulness of searching based class-, attribute- and operation names, students stated that the search based on class, attribute and operation names is very useful. From their comments, they said it was easy to find relevant examples and specific information that they need. They stated that this kind of search for models could answer many questions in their mind, for examples: what should this class look like? How many attributes and operations should be present in the class? How could these relationships be used? Etc.
"It is essential. A very useful feature. I think it is easy to find relevant diagrams when searching for class names"

For easy and useful aspects that student find in the repository or they prefer to be available, most of the students stated that searching is the easiest and most useful aspect of the repository, and they do not prefer other kinds of searching mechanisms at the moment. Other students suggested searching by category or title of class diagrams. Also, some students suggested searching on the description of models. One student suggested searching by relationship name to find how people use it.
"Searching for relevant diagrams based on the classes in my diagram"
"I'd like to be able to search on the description of a model" "Of course we can come up with many different criteria to be added to the repository, but the combination of the class specifications with

the design metrics properties is more than enough in order to find a desirable class diagram"

For which information from class diagrams they found in the repository they use to make their own design, students used the repository to find the following:

- Syntax: *"what a diagram should graphically look like, and how to use class diagram notation"*
- Naming: *"Naming, specific names of classes, operations"*
- Relationships: *"Relations, I got some inspiration for making correction to my class diagram relation after I saw the examples in the repository"*
- Roles and structures: *"Relations, roles and structure"*, *"operations and how to connect entities"*

For the time students spent using the repository, 86% of the students use the repository for 20 30 minutes maximum, which could be considered as a short period compared to two hours experiment time. 76% of students of RG spent less than 40 minutes using the repository. On the other hand, 95% of students of CG spent less than 40 minutes. In the CG group, 30% of the students used the repository for less than 10 minutes. So we conclude that students spent less time using examples when they use it only for updating their models in comparison with using it for creating and updating.

8.6 Threats to Validity

This section discusses the threats to validity.

8.6.1 Internal Validity

We ensured that students are familiar with UML class diagrams, UML assessment, and the UML repository before the experiment. They had the experiment at the end of the software engineering course, and they had a trial two weeks before the experiment.

To avoid participants' expectations from biasing the results, we did not inform them about our experimental hypotheses nor what results we are aiming to know. Before conducting the experiment, the students did not know whether they belonged to RG or CG. We assigned students randomly to RG and CG. We gave all students the same amount of time to finish the task of creating and improving their models.

8.6.2 External Validity

There are two main concern: participants and materials used. Regarding participants, the goal of our research is to study how to learn creating software designs. For this students are an appropriate population. We cannot know if the same finding holds for professionals.

Regarding materials used, the task we gave participants is quite typical in size and complexity compare to the assignments found in UML textbooks. This was confirmed by several lecturers in this field. Also, the modeling task is in a domain that is familiar to students. We cannot know if our findings apply to larger systems.

Models in the repository are related to a wide range of different application domains and vary in size and complexity.

8.7 Conclusion and Future Work

We presented a controlled experiment that aimed at evaluating the effects of using examples for teaching students software design. The main result of our experiment is that model examples aid students in constructing and improving their models. Students appreciated this approach, and they also like the idea of using the repository for finding relevant examples which assist them during the construction of their design. Experts ranked the models produced by the students in the repository group higher than those of the control group for all quality attributes. Students, who were given access to the repository to improve their solution, increased the quality of their solution as assessed by experts for all quality attributes. We observe that using examples makes students more confident in their models: 27% of the repository group see no need to improve their models compared to 16% of the control group. Most of the students stated that using examples is helpful for them in order to create and improve their designs. Students used examples to understand the relationships between classes, naming of class diagram elements, and roles and structure of the class diagram. The model repository is a suitable environment for offering examples. Students preferred the repository over using internet search engines. The main reason of that preference is that it allows them to search for models based on model-contents e.g. class names, which is not possible via internet search.

The time spent for searching examples in the UML Repository is less when students use it only for updating and improving their models. Students suggested some features that could be added to the repository, for example having a model/system name and the possibility to search for models by category and their descriptions.

For future work, we want to study the changes made by the students to improve their models, and then study what they learned from examples. We are going to enrich the repository with some of the features suggested by students. In addition, we think about other useful features e.g. models comparison and plagiarism detection.

Chapter 9

Conclusion and Future Work

In this chapter, we describe the conclusion of this research and outline the future work.

9.1 Conclusion

Our work can be positioned in the area of quality assurance for software designs. Our first effort explored the use of ontologies in assessing the severity of defects. Here we found that our automated prediction method MAPDESO performs well compared with the manual (original) classifications of the defects obtained from the conducted case studies. In addition, MAPDESO performs better than the classifiers in WEKA. We notice that our method is very practical because it is based on IEEE standard [25] for the defects' attributes and its values.

Subsequently, we explored the creation and use of a corpus of UML models (esp. class diagrams). In this thesis, we describe the design of a database which can be accessed via a web-based system. For collecting our corpus, we developed new techniques for classifying class diagrams and extracting class models from images.

We describe some basics characteristics of our corpus. We find that class diagrams are in general not so large, and it can be part of larger systems, but then the model of such a large system is split up across multiple diagrams. We investigate relations between models design metrics, and we find there are many interesting relations between diagrams design metrics, which affects the quality of these diagrams such as the relation between diagrams size and maximum coupling.

We went on to illustrate the usefulness of the corpus through additional studies. We showed how the use of a corpus can be an aid in teaching students how to design class diagrams. We used the corpus to study whether students are reliable assessors of UML class diagram designs from peer-students. The finding is that students evaluation are higher than experts. However, from the quantitative analysis, we found that the students are not eligible to evaluate other students' class diagrams, we perceive from the qualitative analysis that their feedback is similar to experts feedback.Therefore, we conclude that feedback from students is valuable and can help them for improving their design.

We showed how the corpus can be used to find projects, and various data related to the projects such as source code, documentation, test cases that may or may not be available for empirical studies.

We studied the relation between the quality of a UML class diagrams design by looking at the anti-patterns and compare this to the quality of the associated source code – also by looking at the anti-patterns in that. This study showed that anti-patterns can be detected in the design, and these anti-patterns transfer to the source code. In addition, we observe that classes in the design that have anti-patterns have more changes and bugs in the implementation. Therefore, anti-patterns in the design have a big impact on the software implementation and software maintenance.

9.2 Future Work

We believe the corpus of UML models that we have collected will be a valuable source for empirical studies in the future. We have only started to explore its characteristics and use. We are working on expanding the corpus throw extracting class diagrams from documents in Word or Pdf format. More meta-data and related data: linking diagrams to source code is desirable. For this, we are working on more advanced crawling of open source repositories. Another refinement would be to automatically separate reverse engineered diagrams from forward designed diagrams. The UML Repository can be extended to include other types of UML diagrams. Therefore, we are going to include other types of diagrams (sequence diagrams and use case and use case).

We believe that applying machine learning for detecting patterns in the corpus is very useful for learning layout of diagrams, learning roles of classes that proposed by Wirfs-Brock [127]. She suggests roles such as: information holder, controller, decider, and user-interface. From this and using machine learning techniques we try to investigate in which combinations do such roles appear in design? Moreover is there a 'grammar' that can generate well designs or recognize violations of good design combinations?

In addition, learning good and bad naming practices is useful and affects quality attributes of designs such as understandability and layout.

For software quality assurance, using UML repository as a benchmark: i.e. find out which ranges of metrics (threshold) are considered reasonable as a function of the size of a diagrams.

We believe that the repository should be supported with a UML CASE tool, and we are establish a link with Web-UML editor with the repository. At the moment, models in the repository can be opened and edited online via Web-UML. At the end, the data that is collected from Web-UML editor such as class diagrams, user tracking, and feedbacks are going to be stored in the repository. This will enrich the data in the repository and enrich experiment and research that can be conducted.

Bibliography

[1] M. Iliev, B. Karasneh, M. R. V. Chaudron, and E. Essenius, "Automated prediction of defect severity based on codifying design knowledge using ontologies," in *1st International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE 2012)*, pp. 7–11, June 2012.

[2] B. Karasneh and M. R. V. Chaudron, "Extracting uml models from images," in *5th International Conference on Computer Science and Information Technology (CSIT2013)*, pp. 169–178, IEEE, 2013.

[3] B. Karasneh and M. R. V. Chaudron, "Img2uml: A system for extracting uml models from images," in *39th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2013)*, pp. 134–137, IEEE, 2013.

[4] B. Karasneh and M. R. V. Chaudron, "Online img2uml repository: An online repository for uml models.," in *Proceedings of the 3rd International Workshop on Experiences and Empirical Studies in Software Modeling (EESSMOD@MoDELS 2013)*, pp. 61–66, 2013.

[5] T. Ho Quang, M. R. V. Chaudron, I. Samúelsson, J. Hjaltason, B. Karasneh, and H. Osman, "Automatic classification of uml class diagrams from images," in *Proceedings 21st Asia-Pacific Software Engineering Conference (APSEC 2014)*, 2014.

[6] B. Karasneh, D. Stikkolorum, E. Larios, and M. R. V. Chaudron, "Quality assessment of uml class diagrams: A study comparing experts and students," in *MoDELS*, 2015.

[7] D. Stikkolorum, T. Ho Ho Quang, B. Karasneh, and M. R. V. Chaudron, "Uncovering students' common difficulties and strategies during a class diagram design process: an online experiment," in *MoDELS*, 2015.

[8] B. Karasneh, R. Jolak, and M. R. V. Chaudron, "Using examples for teaching software design," in *Proceedings of the 22st Asia-Pacific Software Engineering Conference (APSEC2015)*, 2015.

[9] B. Karasneh, M. R. V. Chaudron, F. Khomh, and Y.-G. Guéhéneuc, "Studying the relation between anti-patterns in models and in source code," in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)*, 2016.

[10] A. Geraci, F. Katki, L. McMonegal, B. Meyer, and H. Porteous, "Ieee standard computer dictionary," *A Compilation of IEEE Standard Computer Glossaries. IEEE Std*, vol. 610, 1991. (cited on pages 9, 13 and 25).

[11] ISO/IEC, "ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models," tech. rep., 2010. (cited on pages 10 and 122).

[12] ISO/IEC, *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001. (cited on page 10).

[13] J. A. McCall, P. K. Richards, and G. F. Walters, "Factors in software quality. volume i. concepts and definitions of software quality," tech. rep., DTIC Document, 1977. (cited on page 10).

[14] B. W. Boehm, J. R. Brown, and M. Lipow, "Quantitative evaluation of software quality," in *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, (Los Alamitos, CA, USA), pp. 592–605, IEEE Computer Society Press, 1976. (cited on page 10).

[15] J. Bansiya and C. Davis, "A hierarchical model for object-oriented design quality assessment," *Software Engineering, IEEE Transactions on*, vol. 28, pp. 4–17, Jan 2002. (cited on page 10).

[16] R. G. Dromey, "A model for software product quality," *IEEE Trans. Softw. Eng.*, vol. 21, pp. 146–162, #feb# 1995. (cited on page 10).

[17] F. Khomh and Y.-G. Guéhéneuc, "Dequalite: Building design-based software quality models," in *Proceedings of the 15th Conference on Pattern Languages of Programs*, PLoP '08, (New York, NY, USA), pp. 2:1–2:7, ACM, 2008. (cited on page 10).

[18] C. Lange and M. Chaudron, "Managing model quality in uml-based software development," in *Software Technology and Engineering Practice, 2005. 13th IEEE International Workshop on*, pp. 7–16, 2005. (cited on page 10).

[19] N. Fenton and J. Bieman, *Software metrics: a rigorous and practical approach*. CRC Press, 2014. (cited on page 11).

[20] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *Software Engineering, IEEE Transactions on*, vol. 20, pp. 476–493, Jun 1994. (cited on page 13).

[21] A. Janes, M. Scotto, W. Pedrycz, B. Russo, M. Stefanovic, and G. Succi, "Identification of defect-prone classes in telecommunication software systems using design metrics," *Information Sciences*, vol. 176, no. 24, pp. 3711 – 3734, 2006. (cited on page 13).

[22] R. Shatnawi and W. Li, "The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process," *Journal of Systems and Software*, vol. 81, no. 11, pp. 1868 – 1882, 2008. (cited on page 13).

[23] Y. Zhou and H. Leung, "Empirical analysis of object-oriented design metrics for predicting high and low severity faults," *Software Engineering, IEEE Transactions on*, vol. 32, pp. 771–789, Oct 2006. (cited on pages 13, 43 and 44).

[24] C. Catal and B. Diri, "A systematic review of software fault prediction studies," *Expert Syst. Appl.*, vol. 36, pp. 7346–7354, #may# 2009. (cited on page 13).

[25] I. Group *et al.*, "1044-2009-ieee standard classification for software anomalies," *IEEE, New York*, 2010. (cited on pages ix, 13, 22, 23, 25, 26, 28, 30, 31, 32, 35, 36, 43, 46, 124 and 155).

[26] T. R. Gruber, "A translation approach to portable ontology specifications," *Knowledge Acquisition*, vol. 5, no. 2, pp. 199 – 220, 1993. (cited on page 14).

[27] N. F. Noy, D. L. McGuinness, *et al.*, *Ontology development 101: A guide to creating your first ontology*, vol. 15. Stanford knowledge systems laboratory technical report KSL-01-05 and Stanford medical informatics technical report SMI-2001-0880, 2001. (cited on pages 14 and 23).

[28] L. Dittmann, T. Rademacher, and S. Zelewski, "Performing fmea using ontologies," in *18th International Workshop on Qualitative Reasoning. Evanston USA*, pp. 209–216, 2004. (cited on page 14).

[29] J. Cardoso, "The semantic web vision: Where are we?," *Intelligent Systems, IEEE*, vol. 22, pp. 84–88, Sept 2007. (cited on pages 15 and 23).

[30] M. Horridge, S. Jupp, G. Moulton, A. Rector, R. Stevens, and C. Wroe, "A practical guide to building owl ontologies using protégé 4 and co-ode tools edition1. 2," *The University of Manchester*, 2009. (cited on pages 15, 16, 23, 24, 25, 28 and 105).

[31] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. E. Lorensen, *et al.*, *Object-oriented modeling and design*, vol. 199. Prentice-hall Englewood Cliffs, 1991. (cited on page 16).

[32] G. Booch, *Object Solutions: Managing the Object-oriented Project*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1995. (cited on page 16).

[33] I. Jacobson, "Object oriented software engineering: a use case driven approach," 1992. (cited on page 16).

[34] D. Milicev, "On the semantics of associations and association ends in uml," *Software Engineering, IEEE Transactions on*, vol. 33, pp. 238–251, April 2007. (cited on pages 16 and 122).

[35] B. Dobing and J. Parsons, "How uml is used," *Communications of the ACM*, vol. 49, no. 5, pp. 109–113, 2006. (cited on page 17).

[36] R. France, J. Bieman, and B. H. Cheng, "Repository for model driven development (remodd)," in *Models in Software Engineering*, pp. 311–317, Springer, 2007. (cited on pages 17 and 79).

[37] R. P. Buse and T. Zimmermann, "Information needs for software development analytics," in *Proceedings of the 34th international conference on software engineering*, pp. 987–996, IEEE Press, 2012. (cited on page 17).

[38] D. Rodriguez, I. Herraiz, and R. Harrison, "On software engineering repositories and their open problems," in *Proceedings of the First International Workshop on Realizing AI Synergies in Software Engineering*, RAISE '12, (Piscataway, NJ, USA), pp. 52–56, IEEE Press, 2012. (cited on pages 19 and 78).

[39] A. Escórcio and J. Cardoso, "Editing tools for ontology creation," *Semantic Web Services: Theory, Tools and Applications; IGI Global (former Idea Group). Hersey, Pennsylvania, USA*, pp. 71–95, 2007. (cited on page 23).

[40] M. Iliev, B. Karasneh, M. Chaudron, and E. Essenius, "Automated prediction of defect severity based on codifying design knowledge using ontologies," in *Realizing Artificial Intelligence Synergies in Software Engineering (RAISE), 2012 First International Workshop on*, pp. 7–11, June 2012. (cited on page 30).

[41] I. H. Witten, E. Frank, and M. A. Hall, *Data Mining: Practical Machine Learning Tools and Techniques*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 3rd ed., 2011. (cited on pages 37 and 59).

[42] I. Witten and E. Frank, "Credibility: Evaluating what's been learned," *Data mining: Practical machine learning tools and techniques*, p. 173, 2005. (cited on page 38).

[43] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pp. 346–355, Sept 2008. (cited on pages 43 and 44).

[44] M. D. Mohamed Suffian, "Defect prediction model for testing phase." Masters thesis, Universiti Teknologi Malaysia, 2009. (cited on pages 43 and 44).

[45] A. M. Hoss, *Ontology-Based Methodology for Error Detection in Software Design*. PhD thesis, 2006. (cited on page 45).

[46] Y. Kalfoglou, *Deploying Ontologies in Software Design*. PhD thesis, Citeseer, 2000. (cited on page 45).

[47] K. Tombre and B. Lamiroy, "Graphics recognition - from re-engineering to retrieval," in *Document Analysis and Recognition, 2003. Proceedings. Seventh International Conference on*, pp. 148–155 vol.1, Aug 2003. (cited on page 50).

[48] L. Fu and L. B. Kara, "From engineering diagrams to engineering models: Visual recognition and applications," *Comput. Aided Des.*, vol. 43, pp. 278–292, #mar# 2011. (cited on pages 50 and 75).

[49] *Image Search Developer's Guide of Google, web-site: https://developers.google.com/image-search/v1/devguide/*. (cited on page 51).

[50] J. Han, M. Kamber, and J. Pei, *Data mining: concepts and techniques: concepts and techniques*. Elsevier, 2011. (cited on page 57).

[51] M. A. Hall, *Correlation-based feature selection for machine learning*. PhD thesis, The University of Waikato, 1999. (cited on page 57).

[52] B. Karasneh and M. R. Chaudron, "Img2uml: A system for extracting uml models from images," in *Software Engineering and Advanced Applications (SEAA), 2013 39th EUROMICRO Conference on*, pp. 134–137, IEEE, 2013. (cited on pages 61 and 137).

[53] B. Karasneh and M. Chaudron, "Extracting uml models from images," in *Computer Science and Information Technology (CSIT), 2013 5th International Conference on*, pp. 169–178, March 2013. (cited on pages 61 and 137).

[54] L. Surhone, M. Tennoe, and S. Henssonow, "Aforge .net: Artificial intelligence, computer vision, .net framework," 2013. (cited on pages 62 and 68).

[55] V. Vashisht, T. Choudhury, and T. Prasad, "Sketch recognition using domain classification," *arXiv preprint arXiv:1211.2742*, 2012. (cited on page 62).

[56] *Sparx Enterprise Architect 11, web-site: http://www.sparxsystems.com/*. (cited on pages 72 and 79).

[57] V. Paradigm, "Visual paradigm for uml," *Visual Paradigm for UML-UML tool for software application development*, 2013. (cited on pages 72 and 79).

[58] M. Lee, H. Kim, J. Kim, and J. Lee, "Staruml 5.0 developer guide," *The Open Source UML/MDA Platform*. (cited on pages 72 and 139).

[59] D. Lucrédio, R. P. d. M. Fortes, and J. Whittle, "Moogle: A model search engine," in *Model Driven Engineering Languages and Systems*, pp. 296–310, Springer, 2008. (cited on page 74).

[60] D. Lu and Q. Weng, "A survey of image classification methods and techniques for improving classification performance," *International journal of Remote sensing*, vol. 28, no. 5, pp. 823–870, 2007. (cited on page 75).

[61] J. A. Shine and D. B. Carr, "A comparison of classification methods for large imagery data sets," *JSM*, pp. 3205–3207, 2002. (cited on page 75).

[62] A. Mishchenko and N. Vassilieva, "Model-based chart image classification," in *Advances in Visual Computing*, pp. 476–485, Springer, 2011. (cited on page 75).

[63] Y. Yu, A. Samal, and S. C. Seth, "A system for recognizing a large class of engineering drawings," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 19, no. 8, pp. 868–890, 1997. (cited on page 75).

[64] B. T. Messmer and H. Bunke, "Automatic learning and recognition of graphical symbols in engineering drawings," in *Graphics recognition methods and applications*, pp. 123–134, Springer, 1996. (cited on page 75).

[65] S. V. Ablameyko and S. Uchida, "Recognition of engineering drawing entities: Review of approaches," *International Journal of Image and Graphics*, vol. 7, no. 04, pp. 709–733, 2007. (cited on page 75).

[66] Q. Chen, J. Grundy, and J. Hosking, "Sumlow: early design-stage sketching of uml diagrams on an e-whiteboard," *Software: Practice and Experience*, vol. 38, no. 9, pp. 961–994, 2008. (cited on page 75).

[67] T. Hammond and R. Davis, "Tahuti: A geometrical sketch recognition system for uml class diagrams," in *ACM SIGGRAPH 2006 Courses*, p. 25, ACM, 2006. (cited on page 75).

[68] E. Lank, J. Thorley, S. Chen, and D. Blostein, "On-line recognition of uml diagrams," in *Document Analysis and Recognition, 2001. Proceedings. Sixth International Conference on*, pp. 356–360, IEEE, 2001. (cited on page 75).

[69] E. Lank, J. S. Thorley, and S. J.-S. Chen, "An interactive system for recognizing hand drawn uml diagrams," in *Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*, p. 7, IBM Press, 2000. (cited on page 75).

[70] L. Qiu, "Sketchuml: The design of a sketch-based tool for uml class diagrams," in *World Conference on Educational Multimedia, Hypermedia and Telecommunications*, vol. 2007, pp. 986–994, 2007. (cited on page 75).

[71] M. Scott, "Wordsmith tools version 7," 2016. (cited on page 78).

[72] J. Di Rocco, D. Di Ruscio, L. Iovino, and A. Pierantonio, "Collaborative repositories in model-driven engineering [software technology]," *IEEE Software*, no. 3, pp. 28–34, 2015. (cited on page 79).

[73] F. Basciani, J. Di Rocco, D. Di Ruscio, L. Iovino, and A. Pierantonio, "Model repositories: Will they become reality ?," in *CloudMDE Workshop at MoDELS 2015, Ottawa, Canada*, 2015. (cited on page 79).

[74] M. Lanza and R. Marinescu, *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007. (cited on page 80).

[75] A. Maraee and M. Balaban, "Efficient recognition of finite satisfiability in uml class diagrams: Strengthening by propagation of disjoint constraints," in *Model-Based Systems Engineering, 2009. MBSE'09. International Conference on*, pp. 1–8, IEEE, 2009. (cited on page 80).

[76] C. Secchi, C. Fantuzzi, and M. Bonfe, "On the use of uml for modeling physical systems," in *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pp. 3990–3995, April 2005. (cited on page 81).

[77] K. C. Thramboulidis, "Using uml in control and automation: a model driven approach," in *Industrial Informatics, 2004. INDIN'04. 2004 2nd IEEE International Conference on*, pp. 587–593, IEEE, 2004. (cited on page 81).

[78] J. Wust, "Sdmetrics: The software design metrics tool for uml," 2014. (cited on page 81).

[79] T. Ho-Quang, M. Chaudron, I. Samuelsson, J. Hjaltason, B. Karasneh, and H. Osman, "Automatic classification of uml class diagrams from images," in *Software Engineering Conference (APSEC), 2014 21st Asia-Pacific*, vol. 1, pp. 399–406, Dec 2014. (cited on page 82).

[80] J. Cohen, *Statistical power analysis for the behavioral sciences*. Academic press, 2013. (cited on page 100).

[81] D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič, "Software fault prediction metrics: A systematic literature review," *Information and Software Technology*, vol. 55, no. 8, pp. 1397 – 1418, 2013. (cited on page 103).

[82] B. Venners, "How to use design patterns. a conversation with erich gamma, part i," *Leading-Edge Java*, 2005. (cited on page 104).

[83] W. H. Brown, R. C. Malveau, and T. J. Mowbray, "Antipatterns: refactoring software, architectures, and projects in crisis," 1998. (cited on page 104).

[84] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *Software Engineering, IEEE Transactions on*, vol. 36, no. 1, pp. 20–36, 2010. (cited on pages 104, 107 and 117).

[85] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y.-G. Guéhéneuc, and E. Aimeur, "Smurf: A svm-based incremental anti-pattern detection approach," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pp. 466–475, IEEE, 2012. (cited on pages 104, 107 and 117).

[86] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 1, pp. 403–414, May 2015. (cited on pages 104, 105 and 117).

[87] S. Vaucher, F. Khomh, N. Moha, and Y.-G. Guéhéneuc, "Tracking design smells: Lessons from a study of god classes," in *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, pp. 145–154, IEEE, 2009. (cited on pages 104, 105, 111 and 117).

[88] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of bad smells in object-oriented code," in *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*, pp. 106–115, Sept 2010. (cited on page 104).

[89] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change-and fault-proneness," *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012. (cited on pages 105 and 117).

[90] Y.-G. Gueheneuc and G. Antoniol, "Demima: A multilayered approach for design pattern identification," *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 667–684, 2008. (cited on page 107).

[91] G. Y-G, H. Sahraoui, and F. Zaidi, "Fingerprinting design patterns," in *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pp. 172–181, Nov 2004. (cited on page 107).

[92] J. Q. Wilson and G. L. Kelling, "Broken windows-the police and neighborhood safety (1982)," *Atlantic Monthly*, vol. 29, p. 31. (cited on page 114).

[93] J. Q. Wilson and G. L. Kelling, "Broken windows-the police and neighborhood safety (1982)," *Atlantic Monthly*, vol. 29, p. 31. (cited on page 116).

[94] R. K. Yin, *Case study research: Design and methods*. Sage publications, 2013. (cited on page 117).

[95] L. Craig, "Applying uml and patterns," *Tredje upplagan, Prentice Hall*, 2002. (cited on page 122).

[96] R. W. Hasker, "Umlgrader: an automated class diagram grader," *Journal of Computing Sciences in Colleges*, vol. 27, no. 1, pp. 47–54, 2011. (cited on page 122).

[97] R. W. Hasker and M. Rowe, "Umlint: Identifying defects in uml diagrams," in *American Society for Engineering Education*, American Society for Engineering Education, 2011. (cited on page 122).

[98] L. B. Christensen, B. Johnson, and L. A. Turner, *Research methods, design, and analysis*. Allyn & Bacon, 2011. (cited on page 123).

[99] D. C. Montgomery, *Design and analysis of experiments*. John Wiley & Sons, 2008. (cited on page 123).

[100] W. F. Tichy, "Hints for reviewing empirical work in software engineering," *Empirical Software Engineering*, vol. 5, no. 4, pp. 309–312, 2000. (cited on page 123).

[101] D. I. Sjøberg, J. E. Hannay, O. Hansen, V. B. Kampenes, A. Karahasanovic, N.-K. Liborg, and A. C. Rekdal, "A survey of controlled experiments in software engineering," *Software Engineering, IEEE Transactions on*, vol. 31, no. 9, pp. 733–753, 2005. (cited on page 123).

[102] J. Boustedt, "Students' different understandings of class diagrams," *Computer Science Education*, vol. 22, no. 1, pp. 29–62, 2012. (cited on page 123).

[103] N. H. Ali, Z. Shukur, and S. Idris, "A design of an assessment system for uml class diagram," in *Computational Science and its Applications, 2007. ICCSA 2007. International Conference on*, pp. 539–546, IEEE, 2007. (cited on page 123).

[104] G. Hoggarth and M. Lockyer, "An automated student diagram assessment system," in *ACM SIGCSE Bulletin*, vol. 30, pp. 122–124, ACM, 1998. (cited on page 123).

[105] S. Kaneda, A. Ida, and T. Sakai, "Understanding of class diagrams based on cognitive linguistics for japanese students," in *Knowledge-Based Software Engineering*, pp. 77–86, Springer, 2014. (cited on page 124).

[106] D. Aguilera, C. Gómez, and A. Olivé, "A complete set of guidelines for naming uml conceptual schema elements," *Data & Knowledge Engineering*, vol. 88, pp. 60–74, 2013. (cited on page 124).

[107] B. Selic, "The pragmatics of model-driven development," *IEEE software*, no. 5, pp. 19–25, 2003. (cited on page 124).

[108] *Supplemental materials for the controlled experiement, 2015.* `http://www.models-db.com/Models2015_SuplementalMaterial.aspx`. (cited on pages 126 and 140).

[109] B. Efron and R. J. Tibshirani, *An introduction to the bootstrap*. CRC press, 1994. (cited on page 126).

[110] I. Spss, "Ibm spss statistics version 21," *Boston, Mass: International Business Machines Corp*, 2012. (cited on page 126).

[111] B. Karasneh and M. R. Chaudron, "Online img2uml repository: An online repository for uml models.," in *EESSMOD@ MoDELS*, pp. 61–66, 2013. (cited on pages 136 and 137).

[112] *UML Repository, web-site: www.models-db.com*, 2013. (cited on pages 136 and 137).

[113] T. Van Gog, L. Kester, and F. Paas, "Effects of worked examples, example-problem, and problem-example pairs on novices' learning," *Contemporary Educational Psychology*, vol. 36, no. 3, pp. 212–218, 2011. (cited on page 137).

[114] R. L. Goldstone and J. Y. Son, "The transfer of scientific principles using concrete and idealized simulations," *The Journal of the Learning Sciences*, vol. 14, no. 1, pp. 69–110, 2005. (cited on page 137).

[115] R. M. Seater, *Core extraction and non-example generation: Debugging and understanding logical models*. PhD thesis, Massachusetts Institute of Technology, 2004. (cited on page 137).

[116] M. L. Gick and K. J. Holyoak, "Schema induction and analogical transfer," *Cognitive psychology*, vol. 15, no. 1, pp. 1–38, 1983. (cited on page 137).

[117] K. Bąk, D. Zayan, K. Czarnecki, M. Antkiewicz, Z. Diskin, A. Wąsowski, and D. Rayside, "Example-driven modeling: model= abstractions+ examples," in *Proceedings of the 2013 International Conference on Software Engineering*, pp. 1273–1276, IEEE Press, 2013. (cited on page 137).

[118] D. Zayan, M. Antkiewicz, and K. Czarnecki, "Effects of using examples on structural model comprehension: a controlled experiment," in *Proceedings of the 36th International Conference on Software Engineering*, pp. 955–966, ACM, 2014. (cited on page 137).

[119] B. Sharif and J. I. Maletic, "An eye tracking study on the effects of layout in understanding the role of design patterns.," in *ICSM*, pp. 1–10, 2010. (cited on page 137).

[120] H. Storrle, "On the impact of layout quality to understanding uml diagrams: Diagram type and expertise," in *Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on*, pp. 49–56, IEEE, 2012. (cited on page 137).

[121] H. Störrle, "On the impact of layout quality to understanding uml diagrams: Size matters," in *Model-Driven Engineering Languages and Systems*, pp. 518–534, Springer, 2014. (cited on page 137).

[122] A. Nugroho, "Level of detail in uml models and its impact on model comprehension: A controlled experiment," *Information and Software Technology*, vol. 51, no. 12, pp. 1670–1685, 2009. (cited on page 137).

[123] N. Juristo and A. M. Moreno, *Basics of software engineering experimentation*. Springer Publishing Company, Incorporated, 2010. (cited on pages 138 and 141).

[124] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika*, pp. 591–611, 1965. (cited on page 141).

[125] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947. (cited on page 141).

[126] R. C. Team, "The r project for statistical computing," *R Foundation for Statistical Computing web-site. www.R-project.org. Accessed June*, vol. 9, 2014. (cited on page 141).

[127] R. Wirfs-Brock and A. McKean, *Object design: roles, responsibilities, and collaborations*. Addison-Wesley Professional, 2003. (cited on page 156).

Summary

In this thesis, we address two problems in Software Engineering. The first problem is how to assess the severity of software defects? The second problem we address is that of studying software designs. We explain each of these problems next in a little more detail.

Assigning severity to a software defect is currently done by software developers based on their experience. In practice it turns out developers do not take the task of registering the severity of defects very seriously and often register just a default value offered by a defect tracking tool. Therefore, we study whether we can provide meaningful automated support for assessing the severity of defects. Automated support for assessing the severity of software defects helps human developers to perform this task more efficiently and more accurately. In this thesis, we present a new approach (MAPDESO) for assessing the severity of software defects based on the IEEE Standard Classification for Software Anomalies. The novelty of the approach lies in its use of uses ontologies and ontology-reasoning which links defects to system level quality properties. The approach was validated by studying how it performs on an industrial project. In this validation, the automated prediction method performs well compared to the manual classification performed on an industrial project. We found that our MAPDESO approach based on ontologies outperforms selected machine learning classifiers. At the company where we conducted the study, people appreciate the result of MAPDESO.

Our research can be positioned in the field of Empirical Software Engineering – this is a branch of Software Engineering that aims to create knowledge through the analysis of artefacts (and processes) that are part of software development projects. The large majority of studies in this field focus on program texts ('source code') expressed in some programming language. While software designs area a critical aspect of most software development projects, software designs are hardly studied by the empirical software engineering community.

One of the main reasons why studying software designs is challenging is the lack of their availability. In our research we found that software designs are commonly stored as UML diagrams in image formats and are available from various sources on the internet. Hence we decided to collect a large number of UML models in images

and convert them to real models. Therefore, we developed the following software tools: 1) UMLCrawler, which can download a large number of UML diagrams from the internet, 2) UMLImgClassifier, which can classify UML class diagrams from other diagrams, and 3) Img2UML, which can extract model information from UML models stored in image formats and generate XMI files. The generated XMIs are compatible with different UML tools, where the models can be edited and analysed. Our validation shows that UMLImgClassifier and Img2UML provide high accuracy for classifying and converting UML diagrams to XMI files.

The aforementioned tools are building blocks for constructing a UML Repository. We built an online repository which contains UML class diagrams, XMIs and various design metrics of the class diagrams. In this thesis we present this repository and some of its services of such as sharing, searching, questioning, ranking, defining experiments, uploading, downloading and charting.

This repository is the first of its kind and we believe it will be a useful resource for the empricial software engineering research community. In support of this, we conducted a series of empirical studies using the UML Repository. These empirical studies are a drop in the ocean of empirical studies that can be conducted using this repository.

Our first study shows some examples of interesting relations between class diagrams design metrics. In this way, the corpus of UML designs can be used to find patterns in UML models and hence to figure out good and bad practices. This yields reference data that can be used in quality assurance of UML designs.

Our next study studies the relation between the quality of the UML design and the associated source code. Our most prominent findings in this study are: First, modeled classes (classes that are both in the design and the source code) undergo more changes and contain more faults than classes that exist only in the source code (hence not in the design). Second, anti-patterns can be detected early in the design, and anti-patterns that exist in the design transfer to the source code in the same classes. Third, modeled classes that have anti-patterns in the design have more changes and faults than other classes that do not have anti-patterns in the design. Our study is the first study of the relation between the quality of UML-based software designs and their impact on the quality of the source code.

Subsequently, we studied how the repository can be used in educational settings. Our first study in this direction investigates the difference between students and experts in assessing the quality of UML class diagrams. We assessed quality along six quality attributes and distinguished between students' self-assessment and peer-assessment. We found that there is a significant difference between experts and students in both self-assessment and peer-assessment. We found a high correlation between experts and peer-assessment for assessing understandability. In addition, we found that there is the single quality attribute understandability which is correlated with most other quality attributes in both experts' assessments and students' peer-assessments. Moreover we found that students seem to avoid giving peers low grades. Hence peer

assessment by students should not be used for grading in class settings. However, using qualitative analysis of the feedback given in conjunction with the assessment, we observed that novices provide comments on the quality of UML designs that is similar to that which experts' give. Therefore, we conclude that the feedback provided by students' is valuable, and while peer-grading may not be a good idea, using students for peer-feedback provides useful information for improving their designs.

After that, we conducted a study into using examples for teaching software design. To this end, we offered students the possibility to use the UML repository while performing some design assignment and for improving an initial design they have made without the repository. We conclude that the examples help students in constructing and improving their design. Quantitative analysis showed that experts graded the models produced by students that used examples higher than students in the control group (without repository) for all quality attributes. We conclude that the model repository is a suitable environment for offering model examples – in the sense that its content is useful, but also that model examples can be found in an effective way. Furthermore, the students appreciate the fact that the UML Repository allows them to search for models based on various parts of the model such as class names, attributes, and operations. This type of search is not available anywhere else, also not via generic search engines on the internet. To conclude, this series of empirical study illustrates the importance of the corpus of UML models.

Samenvatting

In dit proefschrift richten we ons twee problemen in de software engineering. Het eerste probleem betreft de vraag hoe de ernst ('severity') van software gebreken ('defects') te beoordelen? Het tweede probleem dat we beschouwen betreft de vraag hoe wij het ontwerp ('design') van software kunnen bestuderen. We leggen elk van deze problemen in iets meer detail uit.

Het toewijzen van een ernst-classificatie aan een software defect wordt momenteel gedaan door software-ontwikkelaars op basis van hun ervaring. In de praktijk blijkt dat ontwikkelaars die taak van het registreren van de ernst van de defects niet erg serieus nemen en vaak een default waarde bevestigen die wordt voorgedragen door een defect tracking tool. Om deze situatie te verbeteren bestuderen we of het mogelijk is om geautomatiseerde ondersteuning voor de beoordeling van de ernst van de defects te ontwikkelen. Geautomatiseerde ondersteuning voor de beoordeling van de ernst van software defects kan menselijke ontwikkelaars helpen om deze taak efficiënter en nauwkeuriger uit te voeren. In dit proefschrift presenteren we een nieuwe aanpak (MAPDESO) voor de beoordeling van de ernst van software defects op basis van de IEEE Standard Classification voor Software Anomalies. De innovatie bij deze aanpak ligt in het gebruik van ontologieën en ontologie-gebaseerde redenering die defects koppelt aan systeem-niveau-kwaliteitseigenschappen. Deze aanpak is gevalideerd door te bestuderen hoe zij presteert op een industrieel project. In deze validatie presteert onze geautomatiseerde voorspellingsmethode goed in vergelijking met de handmatige toekenning. De MAPDESO aanpak op basis van ontologieën presteerde beter dan enkele geselecteerde machine learning classifiers. Bij het bedrijf waar we de studie uitgevoerd hebben, waardeeren de engineering de resultaten van MAPDESO.

Ons onderzoek kan gepositioneerd worden in het gebied van de empirische Software Engineering - dit is een tak van de Software Engineering die tot doel heeft kennis te creëren door middel van de analyse van producten en processen die deel uitmaken van software development projecten. Bij een grote meerderheid van studies in dit gebied ligt de nadruk op analyse van programmatekst ('source code') - uitgedrukt in een bepaalde programmeertaal. Hiermee wordt voorbij gegaan aan het feit dat het ontwerp van een software systeem een cruciale rol speelt bij van de meeste software development projecten. Echter de ontwerpen van software systemen worden nauwelijks

bestudeerd door de empirische software engineering gemeenschap.

Een van de belangrijkste redenen waarom het bestuderen van software ontwerpen een uitdaging is, is hun gebrekkige beschikbaarheid. In ons onderzoek hebben we vastgesteld dat software ontwerpen vaak worden gearchiveerd als UML diagrammen in 'image'-formaten en dat software ontwerpen in deze vorm gevonden kunnen worden bij verschillende bronnen op het internet. Om deze reden hebben we besloten om een een poging to doen om een corpus van UML-modellen in image-formats van internet te verzamelen en deze te converteren naar echte modellen. Om dit mogelijk te maken ontwikkelden we de volgende software-instrumenten: 1) UMLCrawler, dit is een software tool die een groot aantal UML diagrammen kan downloaden van het internet, 2) UMLImgClassifier, dit is een automatische classifier die UML class diagrammen kan onderscheiden van andere diagrammen, en 3) Img2UML, dit is een software tool die software-model informatie kan extraheren uit UML-images en ze omzet naar XMI bestanden. De gegenereerde XMI-files zijn compatibel met verschillende UML-tools waarmee de modellen kunnen worden bewerkt en geanalyseerd. Onze validatie toont aan dat UMLImgClassifier en Img2UML een hoge nauwkeurigheid behalen bij het classificeren en omzetten van UML diagrammen naar XMI-model bestanden.

De hiervoor genoemde software tools zijn essentiele ingrediënten voor de bouw van een UML Repository: Tijdens dit onderzoek construeerden we een online repository die UML class diagrammen, XMI-files en enkele bij de ontwerpen behorende metrieken bevat. In dit proefschrift presenteren we deze repository en enkele van haar functies zoals het delen, zoeken, ranken, definiëren van experimenten, uploaden, downloaden en genereren van grafieken.

Deze repository is de eerste corpus in zijn soort en we geloven dat het een nuttige bron zal zijn voor de empirische software engineering gemeenschap. Ter ondersteuning van deze stelling, voerden we een reeks van empirische studies met behulp van deze UML Repository uit. Deze empirische studies zijn slechts een druppel in de oceaan van empirische studies die kunnen worden uitgevoerd met deze repository.

Onze eerste studie toont enkele voorbeelden van interessante relaties tussen klassendiagrammen en software-ontwerp metrieken. Op basis van dergelijke relaties kan het corpus van UML ontwerpen worden gebruikt om patronen in UML-modellen te vinden en er achter komen welke goede en slechte praktijken gangbaar zijn. Dit levert referentie-gegevens die gebruikt kunnen worden bij de kwaliteitsborging van UML ontwerpen.

Onze volgende studie onderzoekt de relatie tussen de kwaliteit van UML ontwerpen en de bijbe-horen¬de broncode. Onze meest prominente bevindingen in deze studie zijn de volgende: Ten eerste, klassen die zowel in het ontwerp en de source code voorkomen ondergaan meer veranderingen en bevatten meer fouten dan klassen die alleen voor komen in de broncode (en dus niet in het ontwerp van de software). Dit is een indicatie dat deze klassen een centrale rol spelen en wellicht van een relatief hoge complexiteit zijn. Ten tweede, we tonen aan dat anti-patterns in een vroeg stadium van software ontwerp geïdentificeerd kunnen worden, en dat anti-patterns die bestaan

in het ontwerp van een software systeem zich vertalen naar anti-patterns bij de corresponderende klassen in de source code. Ten derde, klassen die onderdeel zijn van anti-patterns in het software ontwerp ondergaan meer veranderingen en bevatten meer fouten dan andere klassen die geen onderdeel zijn van anti-patronen in het ontwerp. Onze studie is de eerste studie naar de relatie tussen de kwaliteit van UML-gebaseerde software ontwerpen en de kwaliteit van de source code.

Een andere studie onderzoekt hoe het corpus gebruikt kan worden in een educatieve setting. Onze eerste studie in deze richting onderzoekt hoe studenten en experts verschillen in de manier waarop ze de kwaliteit van UML class diagrammen beoordelen. In deze studie maken we een extra onderscheid tussen een self-assessment en peer-assessment van studenten. We vonden dat er een significant verschil is tussen experts en studenten (zowel hun self-assessment als peer-assessment). We vonden een hoge correlatie tussen experts en peer-assessment van studenten voor de beoordeling van begrijpelijkheid van ontwerpen. We vonden dat enkel het kwaliteitsattribuut begrijpelijkheid is gecorreleerd met de meeste andere kwaliteitskenmerken (zowel voor experts en studenten peer-assessments). Een patroon in de data van de studenten lijkt er op te wijzen dat zij vermijden om onvoldoende beoordelingen te geven aan mede-studenten ('peers'). Een implicatie is dat peer-assessment door student niet zonder meer gebruikt kan worden als beoordelings¬mechanisme in de klas. De studenten werd ook gevraagd om tekstuele feedback te geven op ontwerpen van peers (in samenhang met de kwantitatieve beoordeling). Een analyse van die feedback laat zien dat studenten opmerkingen geven over de kwaliteit van de UML ontwerpen die inhoudelijk gezien vergelijkbaar is met de feedback die experts geven. Uit het voorgaande concluderen we dat peer-feedback van studenten nuttig is, terwijl peer-beoordeling door studenten geen goed idee is.

De laatste studie in educatieve setting onderzoekt het nut van voorbeelden bij het onderwijzen van het ontwerpen van software. Hiertoe boden we studenten de mogelijkheid om de UML repository te gebruiken tijdens het uitvoeren van een aantal opdrachten voor het ontwerpen en verbeteren van een software ontwerp (dat in eerste instantie zonder voorbeelden was gemaakt). We concluderen dat de voorbeelden de studenten helpen bij het creëren en het verbeteren van hun ontwerpen. Een kwantitatieve analyse toonde aan dat experts een hogere beoordeling gaven aan de modellen geproduceerd door studenten die gebruik maken van voorbeelden uit de repository dan aan studenten in de controlegroep (zonder gebruik van voorbeelden uit de repository). Deze hogere beoordeling werd gevonden voor alle kwaliteitsattributen van een software ontwerp. We concluderen dat de model repository een geschikte manier is voor het aanbieden van voorbeelden van software designs – niet alleen in de zin dat de inhoud ervan nuttig is, maar ook dat model -voorbeelden op een efficiente manier te vinden zijn. Bovendien blijkt dat studenten waarderen dat de UML Repository hen in staat stelt om te zoeken naar software ontwerpen op basis van verschillende onderdelen van het ontwerp zoals klasse-namen, attributen en operaties. Deze manier van zoeken is nergens anders beschikbaar, ook niet via generieke zoekmachines op het internet.

Deze serie van empirische onderzoeken illustreert het belang en de mogelijkheden van de corpus van UML modellen van software ontwerpen.

About the Author

Bilal Karasneh was born on July 28, 1982 in Irbid, Jordan. He graduated his B.Sc. in computer science from Al al-Bayt University, Jordan, in 2004. He received his M.Sc. in 2009, and the master thesis was entitled "Enhancing 802.11 via signal strength-based data transmission. Since 2004 until 2011, he worked as a teacher in the ministry of education, Jordan. He got a scholarship from Erasmus Mundus program (JOSYLEEN) for pursuing Ph.D. studies. From July 2011, he worked as a Ph.D. candidate at the Leiden Institute of Advanced Computer Science (LIACS), Leiden University, the Netherlands. He worked within the Software Engineering Group under the supervision of Prof. Dr. Michel R. V. Chaudron. Since 2013, he is a visitor researcher at Chalmers University, Sweden. His research interests include quality of software design, UML, software defects, software maintenance, Database management, image analysis and Ontologies – Semantic Web.