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

Karasneh, B.H.A.

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

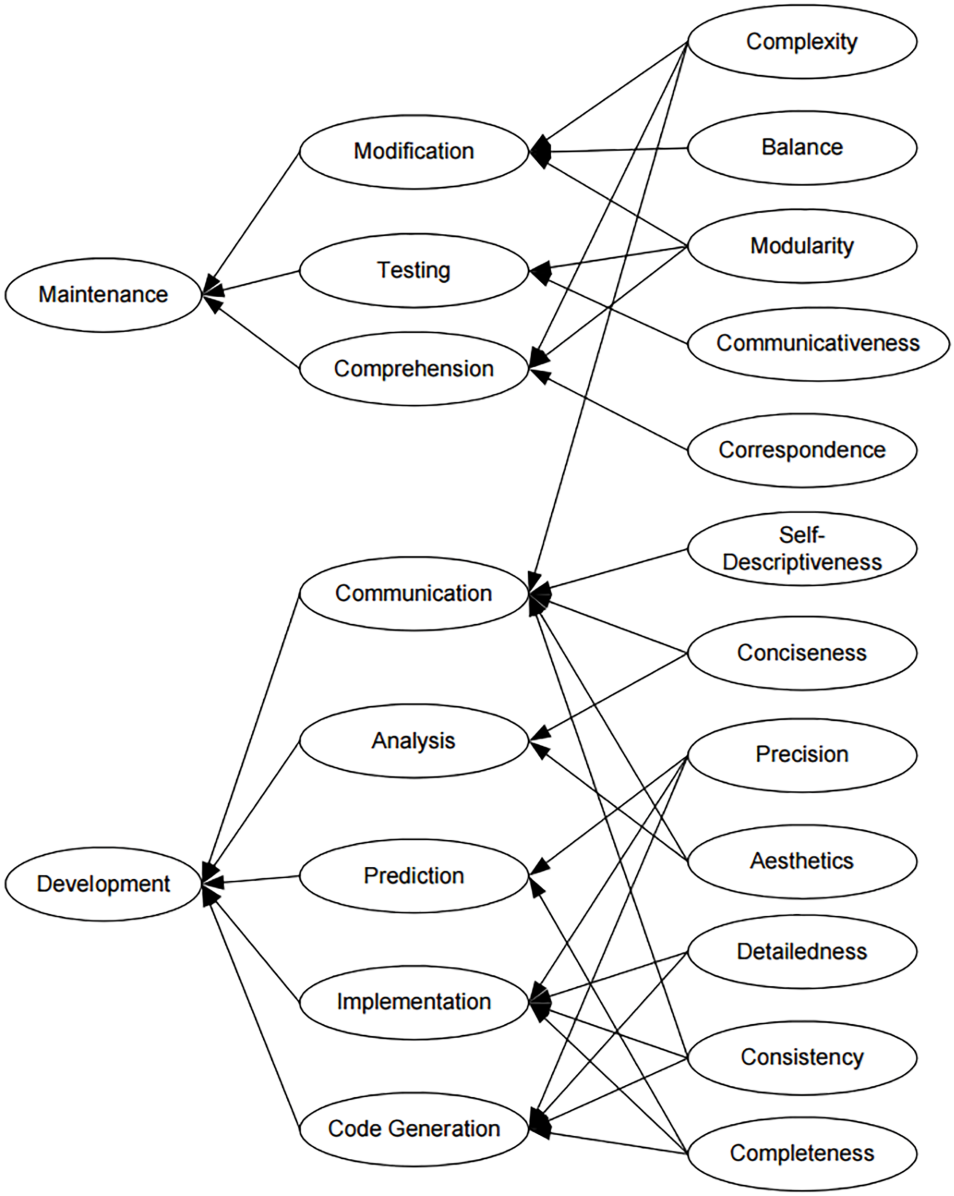


**Figure 2.2:** *Framework of (Lange and Chaudron) for quality of UML models*

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

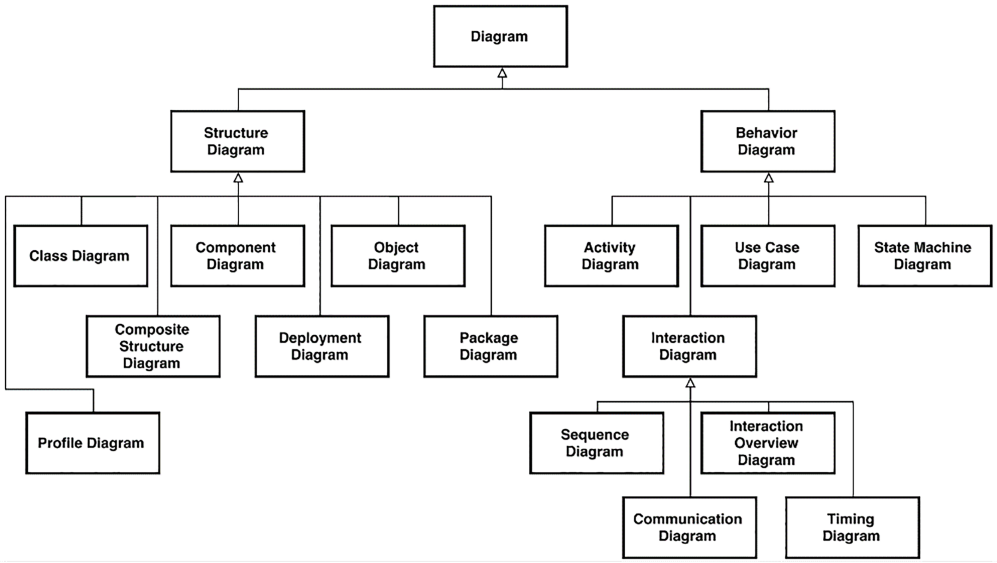


**Figure 2.3:** *The Taxonomy of UML Diagram Types*

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/