Internal Report 2012-04

Universiteit Leiden

Computer Science

A Method for Automated Prediction of Defect Severity Using Ontologies

Name: Martin Pavlov ILIEV
Student-no: s1053574

Date: 10/07/2012

1st supervisor: Dr. M.R.V. (Michel) Chaudron
2nd supervisor: Dr. P.W.H. (Peter) van der Putten

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

A Method for Automated Prediction of Defect Severity Using Ontologies

Martin Pavlov ILIEV

Master's Thesis

LIACS, Leiden University

Logica Netherlands B. V.

July 10, 2012

Universiteit Leiden

logica

be brilliant together

ABSTRACT

Assessing severity of software defects is essential for prioritizing fixing activities as well as for assessing whether the quality level of a software system is good enough for release. In practice, filling out defect reports is done manually and developers routinely fill out default values for the severity levels. Moreover, external factors are a reason for assigning wrong severity levels to defects. The purpose of this research is to automate the prediction of defect severity. We have researched how this severity prediction can be achieved through incorporating knowledge of the software development process using ontologies. In addition, we also employ an IEEE standard to create a uniform framework for the attributes of the defects.

The thesis presents MAPDESO – a Method for Automated Prediction of DEfect Severity using Ontologies. It was developed using industrial case studies during an internship at Logica Netherlands B. V. 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. The results from its validation and comparison with the Weka machine learning workbench indicate that MAPDESO is a good predictor for defect severity levels and it can be especially useful for medium-to-large projects with many defects.

ACKNOWLEDGEMENTS

I would first and foremost like to thank my academic supervisors, Dr. Michel Chaudron and Dr. Peter van der Putten, and my mentor at Logica, ir. Edwin Essenius, for their time, effort, guidance and invaluable feedback throughout the research.

Big thanks to the people at Logica for being very helpful and open-minded and for allowing me to work on industrial case studies. I also want to thank the members of the Software Engineering Group at LIACS for their useful insights and advice throughout the research.

Finally, I would like to express my deepest gratitude to my family since without them none of this would have been possible. I would like to thank them for their inexhaustible support, for their patience and understanding and for encouraging me to always do the very best I can.

Thank you!

CONTENTS

1. INTRODUCTION

Software goes through a testing phase, which aims to find the problems users might experience before the software goes into actual use. The goal of finding these problems is to remove them before the actual use of the software so that the users will not be hindered by them. According to the IEEE Standard Classification for Software Anomalies [2], the cause of a software problem is called a software defect. In order to remove the problems, the defects need to be fixed. Within this thesis well established standards (including the IEEE Standard Computer Dictionary [1] and the IEEE Standard in [2]) are used for defining the necessary terms. The classification in [2] defines a defect as:

- a fault if it is encountered during software execution (thus causing a failure);
- not a fault if it is detected by inspection or static analysis and removed prior to executing the software.

In [1] a fault is defined as an incorrect step, process, or data definition in a computer program, while a failure represents the inability of a system or component to perform its required functions within specified performance requirements. The dictionary relates all these terms to one another by distinguishing between a human action (a mistake), its manifestation (a hardware or software fault), the result of the fault (a failure), and 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.

All users would like to have quality software products. Quality, as given in [1], represents the degree to which a system, component, or process meets specified requirements, customer or user needs or expectations. Therefore, a step towards deploying a high quality software product is to test it first. This is achieved during the system testing phase of the software development life cycle when testing is conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements [1]. This phase results in finding defects in the software product. In general, it is very difficult (or not possible at all) to fix all defects before the deployment date. Hence, these defects need to be categorized so that only the important ones are fixed within the specified time constraint in order to release a quality software product.

1.1. Problem statement

Out of all defects found during the testing phase, the important ones have to be fixed before a predefined deadline. Therefore, a software team needs to decide on the order in which to fix these defects. It is a common practice to assign severity levels to the defects to differentiate between their impacts on the software. The severities of defects represent the different levels of negative impact a defect will have on the deployment of a software product. For example, a severity level *showstopper* is assigned to defects that prevent the release of the software system and immediate attention is required. On the other hand, a severity level *minor* is

assigned to defects that do not prevent the release of the system but the users are annoyed by their presence. It is clear, then, that defects must be assigned the correct severity levels.

The assignment of severity levels to defects is specific for every software system or company and is done manually, usually by test analysts according to their expertise. Different software projects define different sets of severities and, hence, assign different severity levels to their defects. This results in software projects using sets of severities containing three, four or five severity levels (sometimes even more). In addition, it is regularly the case that a defect is assigned the default severity level, which typically is medium. If, at this point, a user is consulted, he/she might not agree with the assignment of default severities and might want some defects to be fixed sooner than others. Moreover, people sometimes make mistakes when assigning severities or are influenced by external factors that lead to assigning wrong severity levels to defects. To address these problems, we have conducted research in the area of how to predict the severity of defects using the knowledge of the software development process while decreasing the workload of the software architects and the test analysts. This means that 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. Its name is MAPDESO – a Method for Automated Prediction of DEfect Severity using Ontologies. Such a method would be especially useful for medium-to-large software systems, which have 100 defects or more.

1.2. Means for achieving the research goal

The means used for achieving the goal of this research are

- the IEEE standard in [2] in order to create a uniform framework for the attributes of the defects, and
- Artificial Intelligence (AI) techniques, namely ontologies, reasoning and automatic classification, in order to capture software defects through an ontology and automatically reason about the defects and their severity levels.

Since different projects define and use different sets of severity levels, creating a uniform framework will be valuable for providing a single set of severity levels that is known to everybody – software architects, developers, test analysts. Therefore, the time and cost for retraining people when they switch projects will decrease since all projects will conform to the same standard for software anomalies.

In addition, researchers have used different techniques to predict the severity levels of defect reports [3], the presence or absence of faults [4] and defects [5], [6]. These techniques include standard text mining methods, logistic regression and machine learning techniques, Six Sigma methodology. Though they have proven to be very useful, they base their results on text mining analysis and on statistical methods. For achieving the research goal we use AI techniques because this way the prediction process is automated and it is based on the different levels of impact defects have on the quality properties of the software (other factors also take part in the prediction process).

Figure 1 presents a summary of everything explained until now. The problems in the figure refer to the problems mentioned in Section 1.1; the method (explained in details in Section 3) includes the means for achieving the goal of this research as given partly in Section 1.1 and in Section 1.2, while the result is the outcome from the method as given in Section 1.1.

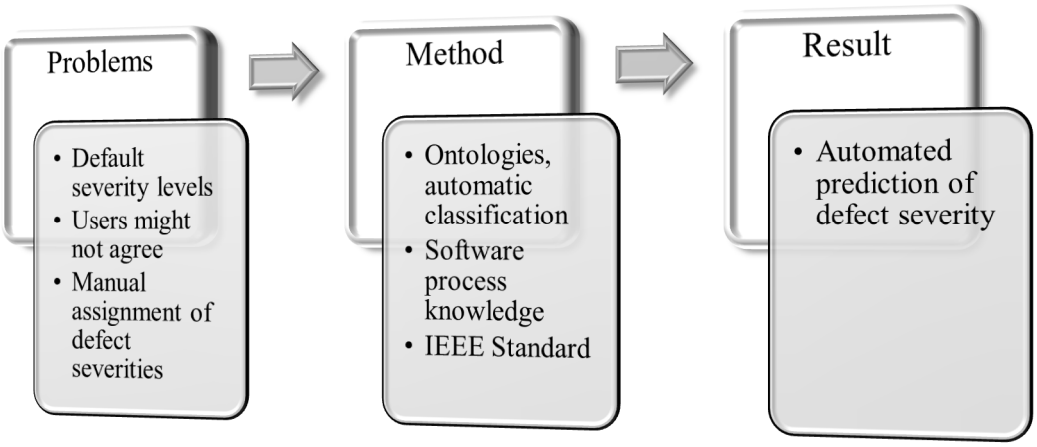

Figure 1. Summary of the current problems, the method to overcome them and the end result.

The next subsection contains details about the above-mentioned IEEE standard while Section 2 presents details about ontologies and ontology engineering.

1.3. IEEE Standard

The IEEE Standard Classification for Software Anomalies (IEEE Std 1044™-2009) [2] is sponsored by the Software & Systems Engineering Standards Committee of the IEEE Computer Society and it was approved on 9 November 2009. This standard provides a uniform approach to the classification of software anomalies, regardless of when they originate or when they are encountered within the software development life cycle. The classification data, given in [2], can be used for a variety of purposes, including defect causal analysis, project management, and software process improvement (e.g., to reduce the likelihood of defect insertion and/or to increase the likelihood of early defect detection). Moreover, the standard 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.

The table in the standard with the most common attributes and their values contains ten attributes. For the purposes of our research the attributes that provide the following information are necessary:

- what is the severity of a defect (only one value is possible),
- what is/are the quality property/properties affected by a defect (one or more values are possible),
- what is/are the type(s) of a defect (one or more values are possible),
- in which phase of the software cycle a defect was inserted (only one value is possible),
- and in which phase of the software cycle a defect was detected (only one value is possible).

Hence, the following five attributes are selected from the standard: Severity, Effect, Type, Insertion activity and Detection activity. The idea behind using specifically them 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.

It should be noted that there are five more attributes – Status, Priority, Probability, Mode and Disposition. They also provide valuable information but it is not essential for the purposes of this research (knowing the values of these attributes for the different defects is not obligatory). Therefore, these five attributes are not considered in the rest of the research.

1.4. Related work

As explained in the previous sections, the severity levels assigned to defects are used to find out what is the impact of that defect on the deployment of the software. It is also known that different software projects assign different severity levels to their defects. More importantly, why a specific defect is assigned one severity and not another and whether both the developers of the software product and its users agree on the assignment of the severity levels are areas that still need more attention.

A new and automated method, which assists the test engineer in assigning severity levels to defect reports, is presented in the paper by Menzies and Marcus [3]. The authors 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 and built to automatically review issue reports and 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 [3] – an automated method for predicting what severity levels to be assigned to defects. However, we base our method on the software development process and software quality properties in order to decide what severity level to assign to a defect so that, in the end, the user satisfaction with the quality of the deployed software product will rise.

Zhou and Leung [4] investigate 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 [3] since 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 AI techniques such as ontologies and automatic classification. This is achieved by developing an ontology and classifying the defects input in it using developed classification rules.

It should also be mentioned that there is research in the area of predicting defects in the design phase [5]. This research resulted in the development of a tool called MetricView. Its goal is to give more insight into UML models by visualizing software metrics that have been computed

by an external tool directly on top of the graphical representation of the UML model. Hence, this tool makes it clearer to see what is correct and what is incorrect in the models. There is also an extension to this tool that provides a lot of additional features such as calculations of metrics within the tool, several views to explore and navigate UML models, visualization of evolution data. The research presented in [5] can be related to ours because we make use of the software process though not for predicting defects but for predicting the severity levels of the defects. An obvious difference is also the fact that we are interested in defects detected mainly from the system testing phase and not from the design phase.

Additional motivation for this work comes from the research conducted by Suffian [6] who establishes a defect prediction model for the testing phase using 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, outlining the possible factors that associate to defect discovery in the testing phase and elaborating on the repeatability and capability of test engineers in finding defects. At the end of his research, 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 and that future effort can focus on improving the defect prediction model to predict defect severity in the testing phase. Therefore, our research represents an extension to the research in [6] since 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). Moreover, our study makes use of defects' attributes as defined in [2] to develop a method that will be applicable to many software projects.

In addition, there is research aimed at the combination of ontologies and software design, which emphasizes on error detection [7], [8]. Such research proves to be very useful since it enhances software design quality, as stated by Hoss [7], and 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 [8]. In this work, we combine ontologies with knowledge of the software development process. We do that in order to automatically predict the severity levels of defects taking into consideration the fact that the defects have already been detected and reported. In other words, our goal is different from the ones mentioned in [7] and [8] though the means to achieve it are similar to some extent.

Another interesting research related to our work is presented in the paper by Jin and Cordy [9]. The authors provide an outline of the design and function of the Ontological Adaptive Service-Sharing Integration System (OASIS). OASIS is a novel approach to integration that makes use of specially constructed, external tool adapters and a domain ontology to facilitate software reengineering tool interoperability through service-sharing. With their work, the authors employ ontologies to facilitate a common and difficult maintenance activity – the integration of existing software components or tools into a consistent and interoperable whole. In our research, we not only employ ontologies for achieving our goals but also consider maintenance activities since our method can be applied to defects detected during maintenance.

1.5. Research contribution

Based on the presented related work, the contribution of this research is the following:

1) We use the knowledge of the software process and quality properties in order to, for example, assign higher severity levels (than originally) to defects inserted during the design and requirements phases. This way, the predicted severity levels will reflect what is important not only according to the developers but also according to the users.
2) We use ontologies and ontology reasoning (AI techniques) to automatically classify the defects input in the ontology into predefined severity levels. To this end, we propose a set of rules that is synthesized based on industrial projects.
3) We use attributes and their values from the well-established IEEE standard in [2] in order to describe the defects and their severity levels in the ontology. This way, the severity assessment method will be applicable to any software project and useful for many people such as software architects, developers, test analysts (under the condition that they will also use that standard).

1.6. Outline

The outline of this thesis is visualized in Fig. 2 and it is organized in the following way. Section 2 presents background information about ontologies, ontology engineering and languages. After that, Section 3 describes the method – MAPDESO, which represents the essence of this work. In Section 4 are presented two industrial case studies with details about how they were conducted and what results were achieved. Then, Section 5 provides the validation of the method. The comparison of the ontology classification with the classification done by an existing software tool is discussed in Section 6. Finally, Section 7 contains the conclusions and the recommendations while Section 8 presents the future work.

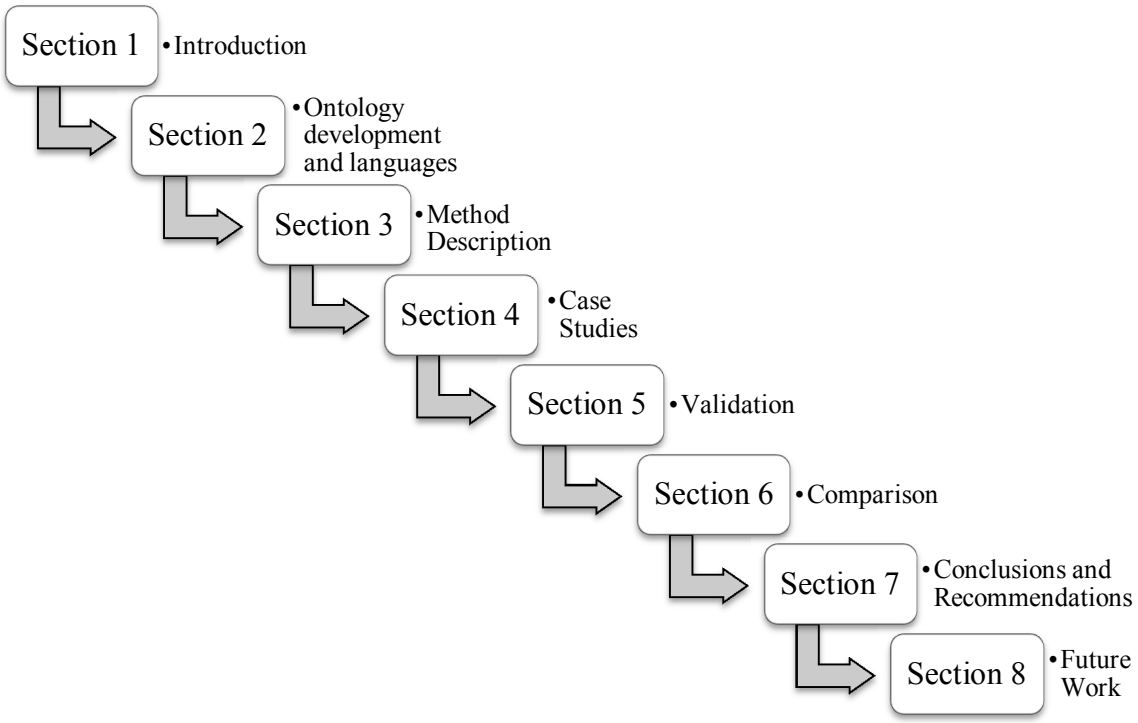

Figure 2. Thesis outline.

2. ONTOLOGY DEVELOPMENT AND LANGUAGES

Software Engineering (SE), as defined in [1], is the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software. Technology innovation demands from SE reliable and sustainable software products. To answer these demands, SE is aided by other computer science disciplines. Artificial Intelligence (AI), for example, is one of these disciplines that will bring SE to further heights. Particularly *ontologies*, an AI concept, will help SE by providing tools (in the forms of ontology languages) and methods that can be used together to facilitate the development of understandable, durable and high-quality software [16].

2.1. Definition

The most common definition of ontologies says that an ontology is an explicit specification of a conceptualization [10]. In other words, ontologies are explicit formal specifications of the terms in the domain and the relations among them [10].

According to a more elaborate version of the definition, an ontology defines a common vocabulary for researchers who need to share information in a domain and it includes machine-interpretable definitions of basic concepts in the domain and relations among them [11]. Many disciplines develop standardized ontologies that domain experts use to share and annotate information in their fields. A particular example for such a discipline is medicine, which has produced large, standardized, structured vocabularies. One of them is the SNOMED ontology, which provides a common vocabulary for clinical terms [11]. Figure 3 visualizes part of the SNOMED ontology. It shows some of the medical conditions caused by epilepsy and their relations to different clinical findings.

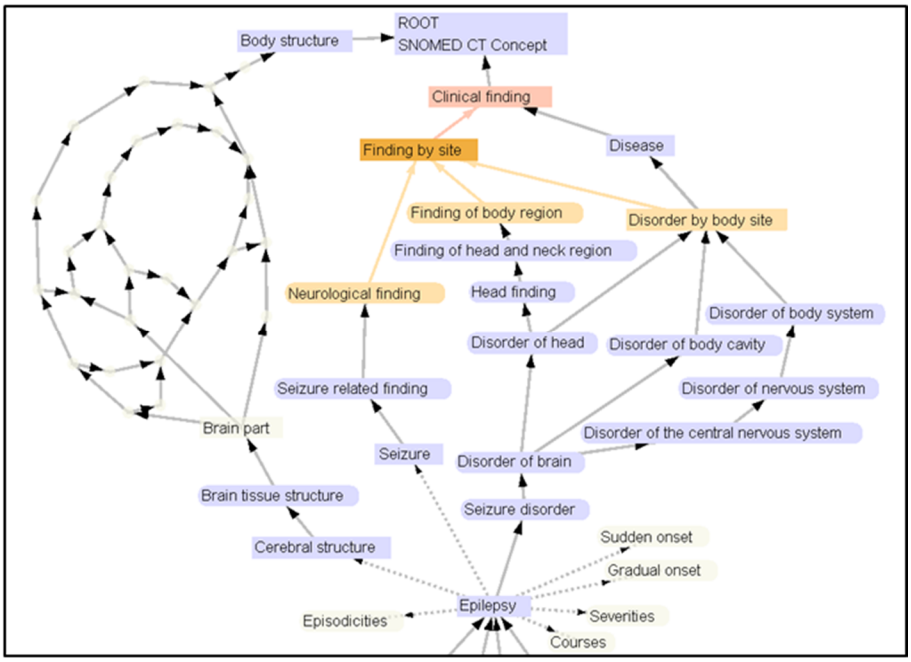

Figure 3. Part of the SNOMED ontology.

In addition, according to [12], an ontology consists of definitions of concepts, relations and rules and is used in knowledge-based systems with the potential to employ inference. Ontologies are also formalized knowledge, represented in a language that supports reasoning.

Once we have defined the term ontology we are ready to dive into the details of ontology development, editors, languages and reasoners.

2.2. Ontology development and editors

Before explaining how to develop ontologies we should answer the question why would someone want to develop an ontology. According to [11], some of the reasons are

- to share 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.

These reasons are complemented by developing an ontology to codify knowledge and to use reasoning to infer new associations from existing ones. An example for this is shown in Fig. 4. On the left is visualized the original hierarchy of the Medical Dictionary for Regulatory Activities (MedDRA). On the right is presented the same hierarchy after ontological reasoning is applied. It is visible that a new association is added between two of the terms – *(PT) Hepatitis Cholestatic* and *(HLT) Hepatitis*, because of their definitions and the existing associations.

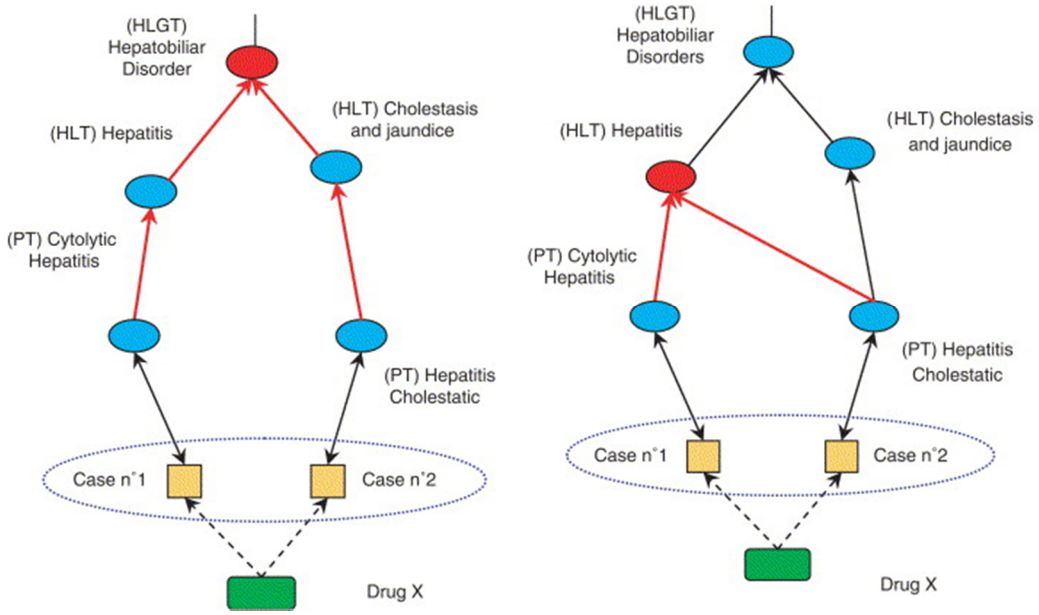

Figure 4. An example for ontological reasoning.

It is often the case that an ontology is not the goal in itself. 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 [11].

It is important to point out that there is no single correct ontology-design methodology [11]. However, a good understanding of ontologies is needed to understand this work. Therefore, below we mention an ontology-design methodology as an example.

An easy and straightforward way to describe the development of an ontology is the top-down approach, as explained in [12]. The goal of this approach is the level of abstraction. There are four levels of abstraction: meta-meta level, meta level, class level and instance level (ordered in top-down fashion). We begin at the meta-meta level and end at the instance level. Of course, every level influences each level of lower abstraction. Therefore, the results from the meta-meta level are used at the meta level and also at the class and instance levels.

The *Meta-meta level* is the phase in which the foundation of the ontology is defined. This includes the decision about the used representation language and a definition of its modeling primitives.

The *Meta level* is the phase in which the key concepts and their relations are defined. This should be done in such a way that the addition of data to an existing ontology should be possible without loss of pre-existing data in that ontology. In other words, the data in an ontology must be preserved.

The *Class level* is used to add more specific descriptions of the knowledge. Having the key concepts already defined in the previous phase, in this phase we define specific sub-concepts of the key concepts.

The *Instance level* is the most specific phase out of the four. The instances represent knowledge that is specific to real projects or systems to which the developed ontology will be applied.

Developing an ontology requires an environment where the above-explained process will be executed. Such environments are called ontology editors. Currently, there are many ontology editors, each having its own strengths and weaknesses. According to the World Wide Web Consortium (W3C)[1], examples of ontology editors are Protégé[2], SWOOP[3], OntoStudio[4] (previously called OntoEdit), NeOn Toolkit[5], Knoodl[6], etc.

Out of the existing ontology editors, Protégé has proven to be the most popular and user-friendly (it is supported by a large community of active users) and the one with many available plug-ins [17], [18]. The results from a survey on Semantic Web practices show that Protégé is the most frequently used ontology editor with a market share of 68.2% [17]. Protégé is ahead of all other editors since the second most frequently used editor is SWOOP with 13.6%, after that is OntoEdit with 12.2% and each of the rest (simple text editor, OntoStudio, etc.) has a share of 10% or less [17].

[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/

In addition to being the most frequently used editor, Protégé is a free, open source ontology editor and knowledge-based framework. It is based on Java, it is extensible and provides a plug-and-play environment. It contains tools to construct domain models and knowledge-based applications with ontologies. Protégé implements a rich set of knowledge-modeling structures and actions that support the creation, visualization and manipulation of ontologies in various representation formats [18]. The Protégé platform supports two main ways of modeling ontologies via the Protégé-Frames[7] and Protégé-OWL[8] editors.

2.3. Web Ontology Language

The most recent development in standard ontology languages is the Web Ontology Language (OWL)[9]. It is endorsed by W3C to promote the Semantic Web[10] vision. OWL is a W3C Recommendation for representing ontologies on the Semantic Web. Moreover, OWL is the language with the strongest impact in the Semantic Web with more than 75% of ontologists selecting this language to develop their ontologies [17].

The Web Ontology Language 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. OWL is based on a logical model, 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 [13].

An ontology that conforms to OWL, called an OWL ontology, consists of classes, properties and individuals, their descriptions and relations. If an OWL ontology is given, the OWL formal semantics specifies how to derive its logical consequences, i.e. facts that are not literally present in the ontology but entailed by the semantics. This can be achieved by using ontology reasoners.

As explained in the OWL Guide[11] and in [13], OWL provides three sublanguages – OWL-Lite, OWL-DL and OWL-Full, 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 can be legally expressed and in what can be validly concluded.

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 [13]. 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 (all

[7] http://protege.stanford.edu/overview/protege-frames.html
[8] http://protege.stanford.edu/overview/protege-owl.html
[9] http://www.w3.org/2004/OWL/
[10] http://www.w3.org/2001/sw/
[11] http://www.w3.org/TR/owl-guide/

entailments are guaranteed to be computed) and decidability (all computations/algorithms will finish in finite time) of reasoning systems. OWL-DL is so named because it is based on *Description Logics* (DL). According to [13], Description Logics represent a decidable fragment of First Order Logic and are amenable to automated reasoning. Therefore, it is possible to automatically compute the classification hierarchy and check for inconsistencies in an ontology that conforms to OWL-DL [13].

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 ontologies, as stated in [13].

2.4. Reasoners

A reasoner (also called inference engine) is a software application that derives new facts or associations from existing information [17]. It is a key component for working with ontologies. The survey results in [17] indicate that the most popular reasoners are Jena[12], RacerPro[13], Pellet[14] and FaCT++[15].

Jena is a Java framework for building Semantic Web applications, as explained on the Jena project's home page. Although it includes an inference engine to perform reasoning, it is a comprehensive toolset. It provides a collection of tools and Java libraries to help programmers develop Semantic Web and linked-data applications, tools and servers.

RacerPro is the commercial name of the RACER software (Renamed ABox and Concept Expression Reasoner). The origins of RacerPro are within the area of Description Logics. It can be used as a system for managing OWL ontologies and it can also be used as a reasoning engine for ontology editors such as Protégé.

Pellet is an open-source Java-based OWL-DL reasoner, which provides standard and cutting-edge reasoning services for OWL ontologies. It supports the full expressivity of OWL-DL and is the first sound (all provable statements are true) and complete (all true statements are provable) DL reasoner that can handle this expressivity. It provides functionalities to check consistency of ontologies, classify the taxonomy (this is the superclass-subclass hierarchy in an ontology), check entailments, etc. Pellet is used in a number of projects, from pure research to industrial settings. According to [14], it has proven to be a very reliable tool for working with OWL-DL ontologies and experimenting with OWL extensions.

FaCT++ is the new generation of the FaCT[16] OWL-DL reasoner. FaCT++ uses the established FaCT algorithms, but with a different internal architecture. Moreover, FaCT++ is implemented using C++ in order to create a more efficient software tool and to maximize portability.

[12] http://jena.apache.org/about_jena/about.html
[13] http://www.racer-systems.com/products/racerpro/
[14] Old link: http://www.mindswap.org/2003/pellet/. New link: http://pellet.owldl.com.
[15] http://owl.man.ac.uk/factplusplus/
[16] http://www.cs.man.ac.uk/~horrocks/FaCT/

2.5. Ontology editor, language and reasoner used in the research

When considering how to use ontologies and ontology reasoning for achieving the goal of this research, we realized that it is imperative to be able to perform automated reasoning on the developed ontology (as already mentioned). This process includes checking the consistency of the ontology and classifying the taxonomy, which in our case means classifying the defects into predefined severity levels according to developed rules. Hence, we made the following choices.

The Protégé platform was selected for the ontology development because of its functionality and popularity, as explained in Section 2.2. Since the ontology language selected was OWL (see Section 2.3) – leading to the development of an OWL ontology, the Protégé-OWL editor was the obvious choice from the Protégé platform.

As explained earlier, the OWL language provides three sublanguages. So, the next question was which sublanguage fits the purposes of our research. It turned out that there are some simple rules of thumb how to find out which sublanguage to use [13]. OWL-Lite was not considered because it provides simple constructs and they are not sufficient. Therefore, the choice had to be made between OWL-DL and OWL-Full. For our research, it is important to be able to carry out automated reasoning on the ontology and, as explained in Section 2.3, OWL-DL provides such a possibility while OWL-Full does not guarantee it. Hence, OWL-DL was chosen as the OWL sublanguage.

Last but not least, we had to choose an appropriate reasoner to use in the ontology. At this point of time, we already knew that the ontology will be developed using Protégé and it will conform to OWL-DL, so the choice of the reasoner was relatively straightforward. As explained in Section 2.4, Pellet supports the full expressivity of OWL-DL and using it we can check the consistency of the ontology and classify the taxonomy. Therefore, we decided to use Pellet as the OWL-DL reasoner in the research.

2.6. Section summary

In this section we provided information about ontologies, ontology development and editors, the Web Ontology Language and popular reasoners. Moreover, we discussed which ontology editor, language and reasoner we used in the research and why. Hence, it is now safe to continue further with the details of the developed method and the case studies in the research.

3. METHOD DESCRIPTION

The end result from the method described in this section – MAPDESO, is the solution to the problems we are facing, as mentioned in Section 1.1 and shown in Fig. 1.

The method 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 [2]). We should point out that the two main purposes to use this ontology, though not the only ones, are

- sharing common understanding of the structure of information among people or software agents – achieved by using the IEEE standard for the defects' attributes and their values;
- enabling reuse of domain knowledge – achieved by reusing the ontology and the developed classification rules in order to predict the severity levels of defects from various projects.

Although there are different reasons for developing and using ontologies, as mentioned in Section 2.2, it turns out that the above two are the most common reasons for using ontologies, as evident by the survey results in [17].

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 using it can be done multiple times. In the first subsection we will describe the process of developing the ontology so that it will be clear later how the ontology and the classification work. After that, we will explain the method and we will refer to using the ontology as a black box process – only the input and the output will be mentioned.

3.1. Developing the ontology

In Section 2.2, we referred to an approach for ontology development. For clarity and understandability, we will follow that approach when explaining how the ontology was developed.

3.1.1. Meta-meta level

This is the phase for defining the foundation of the ontology. In fact, we have completed this phase since we already know which ontology editor, language and reasoner we will use – Protégé-OWL, OWL-DL and Pellet, respectively (for more information refer to Sections 2.2-2.5). Hence, the ontology development approach has a predefined meta-meta level.

3.1.2. 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 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 [11]. 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 [13]. 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 in [2]. 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. 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. Its values are the activities during which a defect is inserted.
- DetectionActivity – this class represents attribute Detection activity from the IEEE standard. Its values are the activities during which a defect is detected.

We created the properties describing the relations between the defects and the attributes from the standard. Hence, these properties describe the relations between class Defect and classes Effect, Type, InsertionActivity and DetectionActivity (if these properties relate to the classes, then the same relations will also hold for the respective subclasses). Figure 5 presents the created classes and properties for the ontology.

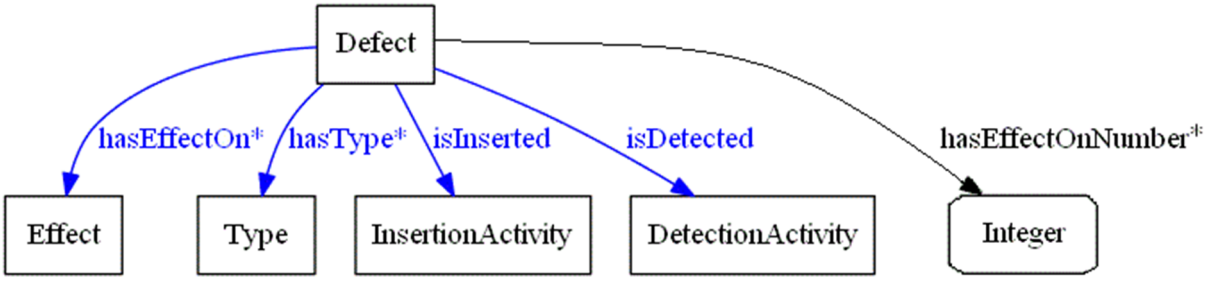

Figure 5. The created classes and properties for the ontology.

On the figure are shown five properties. They are the following:

hasEffectOn* – this is an object property (linking an individual to an individual [13]) that relates class Defect (domain of the property) to class Effect (its range). Hence, this property relates a defect to one or more quality properties (e.g., performance, functionality) affected by it. The asterisk at the end of the property means that its range accepts one or more values.

hasType* – this is an object property that relates class Defect (domain) to class Type (range). This property relates a defect to one or more values of the range (e.g., data, interface). The asterisk at the end of the property means that its range accepts one or more values.

isInserted – this is an object property that relates class Defect (domain) to class InsertionActivity (range). In other words, this property relates a defect to its insertion activity

(e.g., design, coding). The absence of an asterisk at the end of the property means its range accepts only a single value, which implies that this is a functional property [13].

isDetected – this is an object property that relates class Defect (domain) to class DetectionActivity (range). In other words, this property relates a defect to its detection activity (e.g., supplier testing, production). The absence of an asterisk at the end of the property means its range accepts only a single value, which, as above, implies this is a functional property [13].

hasEffectOnNumber* – this is a datatype property (linking an individual to a specific datatype [13], 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 at the end of the property means that its range accepts one or more values (the integer 1 or the integer 2, etc.).

It is important to note that the relations defined by the five properties will be used specifically for the subclasses of the classes mentioned above. Moreover, in Fig. 5, the object properties are depicted in blue color while the datatype property is depicted in black for easier differentiation. The datatype Integer is given in a rounded rectangle to point out that it is not a class (depicted with rectangles) but a datatype.

3.1.3. Class level

In this phase we define the sub-concepts of the key concepts defined at the Meta level. In the current ontology, this means that we will add the required subclasses to the existing classes. Hence, for class Defect, we defined six subclasses. They are the following:

- DefectID – this class represents all defects input in the ontology as its subclasses.
- DefectWithBlockingSL – this class represents all defects assigned blocking severity level (they are displayed as its subclasses after performing the ontology classification).
- DefectWithCriticalSL – this class represents all defects assigned critical severity level (they are displayed as its subclasses after performing the ontology classification).
- DefectWithMajorSL – this class represents all defects assigned major severity level (they are displayed as its subclasses after performing the ontology classification).
- DefectWithMinorSL – this class represents all defects assigned minor severity level (they are displayed as its subclasses after performing the ontology classification).
- DefectWithInconseqSL – this class represents all defects assigned inconsequential severity level (displayed as its subclasses after performing the ontology classification).

The five classes that are related to the five severity levels from the IEEE standard are defined as disjoint from each other because every defect is assigned one and only one severity level.

In the ontology class hierarchy the defects are subclasses of class DefectID. As mentioned earlier and clearly stated in [13], one of the key features of OWL-DL is that the superclass-subclass relationships can be computed automatically by a reasoner. Hence, to use this feature, we have to input the specific defects as separate classes. Moreover, they are input as subclasses of class DefectID for clarity and readability of the overall ontology taxonomy.

Figure 6 presents the class hierarchy for class Defect and its subclasses.

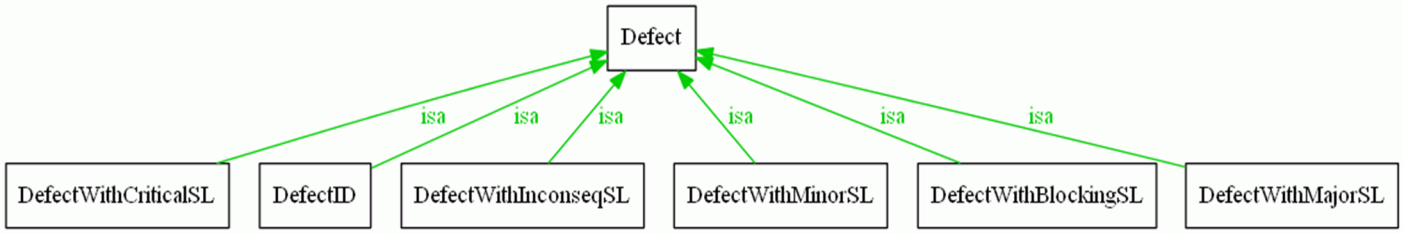

Figure 6. Class Defect and its subclasses.

Next, we created the subclasses for the other four classes. Since the classes are the attributes from the IEEE standard, their subclasses are the values of the respective attributes. For clarification and readability, Fig. 7 shows class Effect together with its relation to the quality properties (the hasvalue arrow). Moreover, the figure shows the created subclasses (using isa arrows) which are the values of attribute Effect. These subclasses are listed below together with their definitions, as taken from the IEEE standard in [2]:

- Functionality – actual or potential cause of failure to correctly perform a required function (or implementation of a function that is not required), including any defect affecting data integrity.
- Usability – actual or potential cause of failure to meet ease of use requirements.
- Security – actual or potential cause of failure to meet security requirements, such as those for authentication, authorization, privacy/confidentiality, accountability (e.g., audit trail or event logging), and so on.
- Performance – actual or potential cause of failure to meet performance requirements (e.g., capacity, computational accuracy, response time, throughput, or availability).
- Serviceability – actual or potential cause of failure to meet requirements for reliability, maintainability, or supportability (e.g., complex design, undocumented code, ambiguous or incomplete error logging).

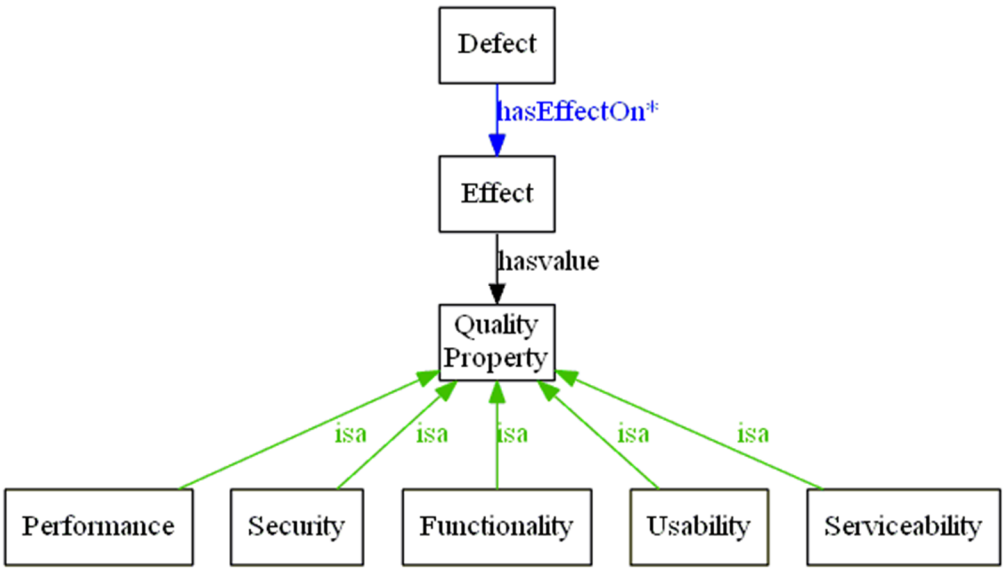

Figure 7. Class Effect and its subclasses – the values of attribute Effect.

Figure 8 presents class Type and its subclasses using isa arrows. These subclasses are the values of attribute Type. They are listed and defined below using the definitions from [2]:

- Data – defect in data definition, initialization, mapping, access, or use, as found in a model, specification, or implementation.
- Interface – defect in specification or implementation of an interface (e.g., between user and machine, between two internal software modules, between software module and database, between internal and external software components, between software and hardware, etc.).
- Logic – defect in decision logic, branching, sequencing, or computational algorithm, as found in natural language specifications or in implementation language.
- Description – defect in description of software or its use, installation, or operation.
- Syntax – nonconformity with the defined rules of a language.
- Standards – nonconformity with a defined standard.
- Other – defect for which there is no defined type.

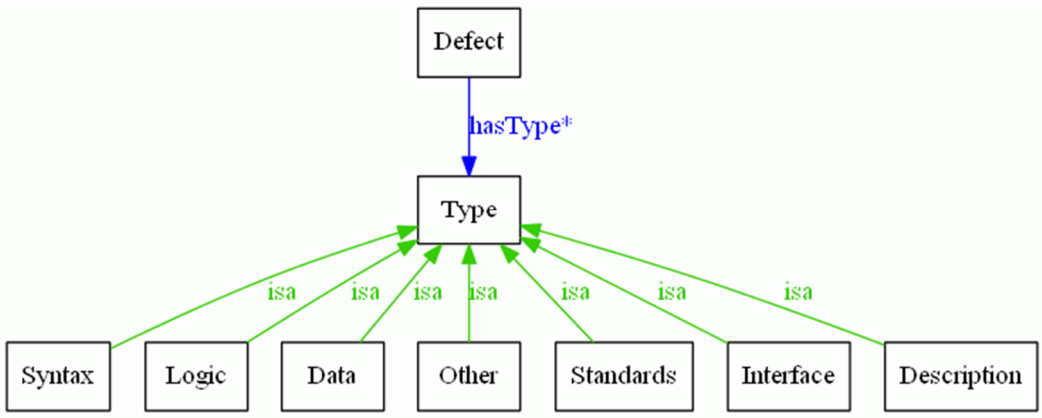

Figure 8. Class Type and its subclasses – the values of attribute Type.

Figure 9 shows class InsertionActivity and its subclasses. The subclasses represent the values of attribute Insertion activity. They are given below together with their definitions from [2]:

- InRequirements – defect inserted during requirements definition activities (e.g., elicitation, analysis, or specification).
- InDesign – defect inserted during design activities.
- InCoding – defect inserted during "coding" or analogous activities.
- InConfiguration – defect inserted during product build or packaging.
- InDocumentation – defect inserted during documentation of instructions for installation or operation.

Figure 10 presents class DetectionActivity and its subclasses, which represent the values of attribute Detection activity. These subclasses and their definitions from [2] are listed below:

- FromRequirements – defect detected during synthesis, inspection, or review of requirements.
- FromDesign – defect detected during synthesis, inspection, or review of design.
- FromCoding – defect detected during synthesis, inspection, or review of source code.

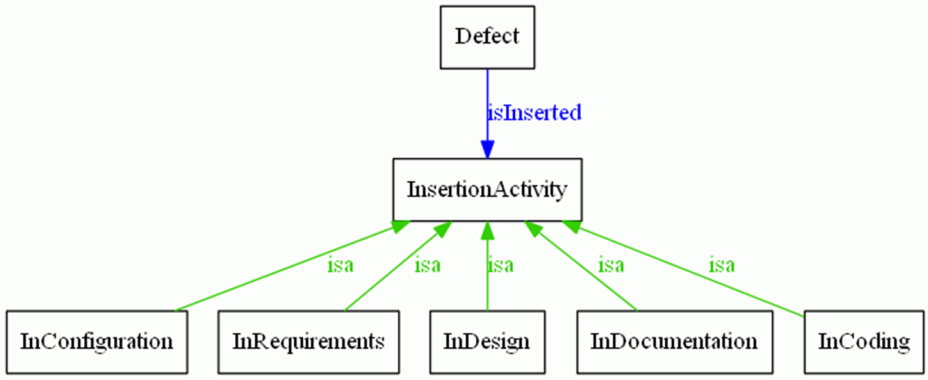

Figure 9. Class InsertionActivity and its subclasses – the values of attribute Insertion activity.

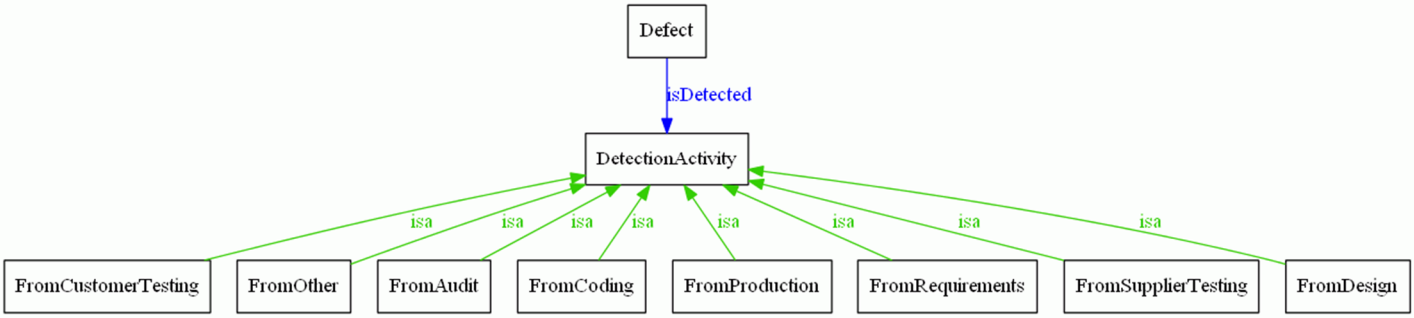

Figure 10. Class DetectionActivity and its subclasses – the values of attribute Detection activity.

- FromSupplierTesting – defect detected during any testing conducted by the supplier.
- FromCustomerTesting – defect detected during testing conducted by the customer.
- FromProduction – defect detected during production operation and use.
- FromAudit – defect detected during an audit (pre-release or post-release).
- FromOther – defect detected during any other activity, such as user/operator training or product demonstrations.

3.1.4. Instance level

This is the most specific phase. Instances represent knowledge that is specific to real projects or systems to which the developed ontology will be applied. Hence, the specific defects input in the current ontology can be regarded as instances. However, as already explained, the defects are input as classes that are subclasses of DefectID. For example, Fig. 11 shows five particular defects input in the ontology as subclasses of DefectID (the other defects are not present because of space concerns). In fact, Fig. 11 represents an extended version of Fig. 6 up to some minor layout differences, which are explained later.

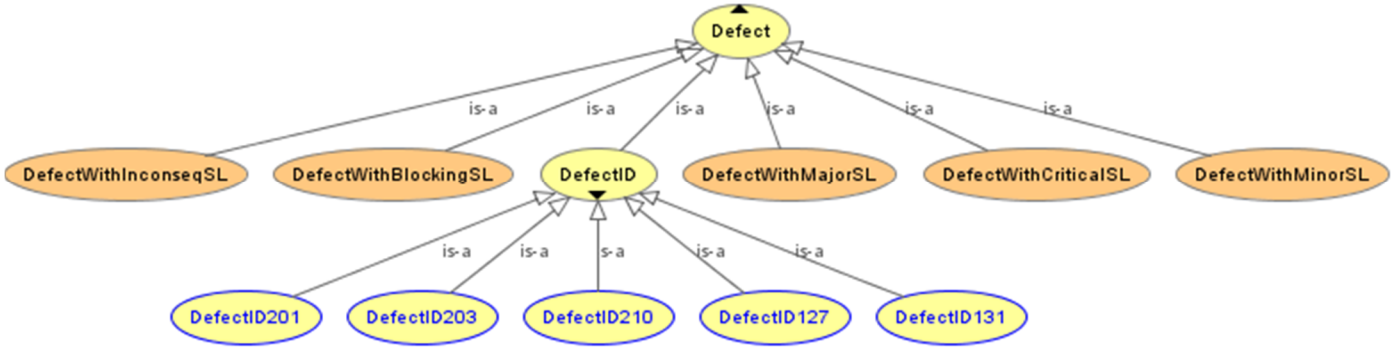

Figure 11. Class Defect, its subclasses and five particular defects.

3.1.5. Classification rules

For the current ontology we have also developed classification rules that are responsible for the classification of the input defects into the five severity levels from the standard. 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 and is assigned the severity level corresponding to the class (i.e., blocking-, critical-, major-, minor- or inconsequential severity level). On the other hand, if a defect belongs to one of the five classes, then it satisfies the set of rules corresponding to that class.

Next, we list the rules for each of the five classes and explain their meaning.

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

These sub-rules mean the following: an entity is assigned blocking severity level if and only if it is: (R1.1) a defect; (R1.2) affecting at least four of the values of attribute Effect (which represent quality properties as already mentioned).

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)

These sub-rules mean the following: an entity is assigned critical severity level if and only if it is: (R2.1) a defect; (R2.2) affecting exactly two or exactly three of the values of attribute Effect; (R2.3) inserted during the design phase or the requirements phase, or inserted during the coding phase or the configuration phase and affecting exactly three values of attribute Effect or at least two values of attribute Type; (R2.4) affecting one or more of the values

Data, Interface or Logic of attribute Type; (R2.5) detected during the coding phase, or the supplier testing phase, or the customer testing phase, or during production use.

In *Rule 2* the operator or represents logical disjunction and the operator and represents logical conjunction. The same applies for these operators in the other rules (if they are present in the other rules).

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

These sub-rules mean the following: an entity is assigned major severity level if and only if it is: (R3.1) a defect; (R3.2) not a defect with blocking severity level, and not a defect with critical severity level or it is inserted during the coding phase or the configuration phase and is affecting exactly two values of attribute Effect and only one of the values Data or Interface or Logic of attribute Type, and not a defect with minor severity level and not a defect with inconsequential severity level (the reason for adding the part of this sub-rule after not DefectWithCriticalSL and before and not DefectWithMinorSL is explained in details in Appendix A in order not to disrupt the flow of the method description).

In *Rule 3* the operator not represents negation (also called logical complement). The same applies for this operator in the other rules (if it is present in the other rules).

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

(R4.2) hasEffectOn some (not Usability and not Security)

(R4.3) hasEffectOn only (not Usability and not Security)

(R4.4) hasEffectOnNumber max 1

These sub-rules mean the following: an entity is assigned minor severity level if and only if it is: (R4.1) a defect; (R4.2) affecting some values of attribute Effect that are not Usability and Security; (R4.3) affecting only values of attribute Effect that are not Usability and Security (this sub-rule is needed in order to make sure that the defect can *only* have the specified values – such a sub-rule is known as a closure axiom [13]); (R4.4) affecting at most one value (and, therefore, exactly one) of attribute Effect.

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

These sub-rules mean the following: an entity is assigned inconsequential severity level if and only if it is: (R5.1) a defect; (R5.2) affecting value Usability of attribute Effect; (R5.3) affecting only value Usability of attribute Effect (as above, this sub-rule is needed in order to make sure that the defect can *only* have the specified value).

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 Case Studies 1 and 2 – see Section 4) and on heuristic strategies, such as intuitive judgment, etc. The rules were later improved to be as general as possible in order to apply the method to various software projects (see the validation in Section 5 for an example).
- use designers' recommendations – the designers' logical argumentation for translating the user requirements into the software design and for assigning severity levels to eventual defects is studied and incorporated in the rules.
- 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. Therefore, more users of the software product will be satisfied.
- 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.

As a result, using these rules, defects will be assigned severity levels in a way that will reflect what is important not only according to the developers/test analysts but also according to the users of the software system.

3.2. The method flow

It is clear now how the ontology was developed. In this subsection we will focus on how to use the ontology in order to automatically predict the severity levels of defects from different projects. The method consists of the following steps: detecting defects; analyzing and converting the information about the defects into the information needed as input for the ontology; entering the converted information about the defects into the ontology; and, lastly, predicting the severity levels of the defects input in the ontology through a single click of a button. These steps are illustrated in the UML activity diagram in Fig. 12 on the next page. The diagram represents a reference point for the description of the whole method flow.

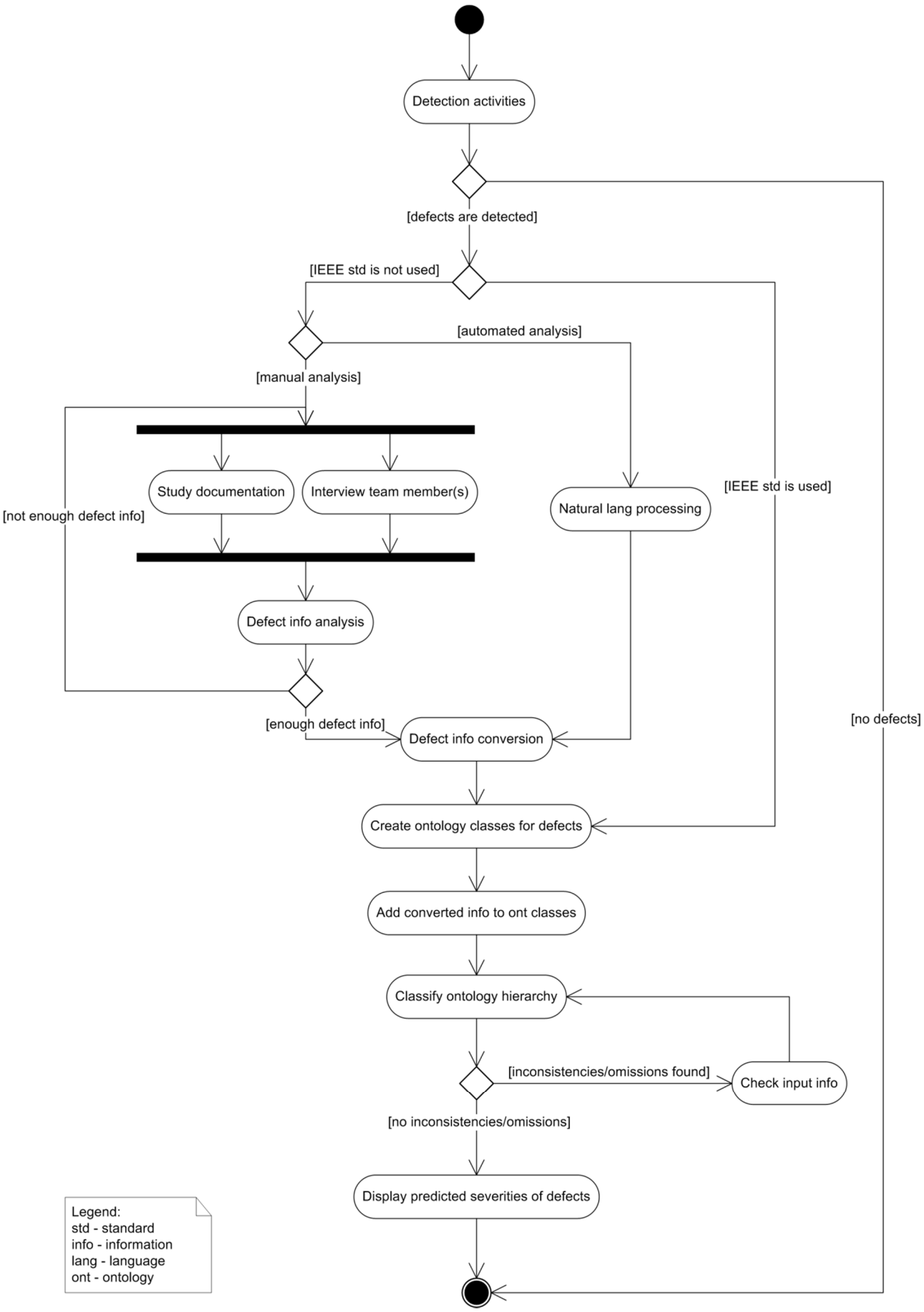

Figure 12. Activity diagram for automated prediction of defects' severity levels.

Before explaining the details of MAPDESO, it should be noted that there are two options when using it, as illustrated in Fig. 12. The first option is to apply the method to a project that does not use the IEEE standard in [2] for describing its defects. And the second option is to apply the method to a project that has adopted the IEEE standard for describing its defects. The obvious difference is the omission of the second step from the method as given above. The reason stems from the fact that once a project is using the IEEE standard 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 input the defects in the ontology. However, below we will describe and explain all steps of the method.

3.2.1. Detection of defects

The testing activity in the software development cycle detects defects, which software teams have to fix. For our research, testing at the system level was the main source of defects. However, we also took into consideration defects detected during the coding phase and during maintenance. Therefore, the four activities, used in this method to detect defects (no detected defects is also a possibility), as defined in [2], are

- coding – defects detected during synthesis, inspection, or review of source code;
- supplier testing – defects detected during testing conducted by the supplier;
- customer testing – defects detected during testing conducted by the customer;
- production – defects detected during production operation and use.

As explained earlier, the next step (Section 3.2.2) is required for projects not using the IEEE standard for describing their defects and it is redundant for projects using the standard for describing their defects.

3.2.2. Analysis and conversion of the defects' information

Taking this step implies that the software project to which the method is applied has not adopted the IEEE standard for describing its defects. Hence, the information about detected defects is gathered and stored in a way that is, most probably, specific only for the project in question. For example, the set of severity levels might contain three, four, five or more levels; the defect tracking system might not contain any information about a defect's insertion activity or type; etc. After analyzing the available information about defects from such a project, it is evident that this information has to be converted into the defect attributes and their values from the standard in [2] in order to apply the method to the project. In our research, the analysis and conversion were done manually. This manual process consisted of:

- studying the project documentation and the available information from the defect tracking system – this way, we were able to understand the information contained in the defect reports;
- conducting interviews with members of the software team – we used their project knowledge in order to gather the information needed for the conversion.

After that, we analyzed the data acquired from the above two steps and if the data were not enough, the steps were repeated. Then, based on the analysis and on the recommendations of

the software team members, we converted the gathered information about the defects into the defect attributes and their values from the standard. The used attributes are Effect, Type, Insertion activity and Detection activity, as defined in [2].

It turned out that it is quite easy to present the results from the analysis and conversion step in a table. The table has five columns representing the Defect ID and the used attributes and every row after the first one (which is the top row) represents separate defects with the respective values of the attributes for each and every defect. Table 1 below shows the top row of the table together with example values of the attributes for a fictitious defect.

Table 1. The format of the table presenting the results from the analysis and conversion step

Defect ID	*Effect*	*Type*	*Insertion Activity*	*Detection Activity*
001	Functionality; usability	Logic	Coding	Supplier testing
…	…	…	…	…

As a future direction, there might be another way to complete the analysis that, in fact, can automate it. It is possible to use natural language processing and data mining algorithms to extract the needed information from the defect reports. This way, the extracted defect information will be converted into the attributes and their values from the standard. Although this option was not used in the research, it might be a very useful way to further automate this method, as explained in Section 8.

3.2.3. Entering the converted information into the ontology

This step presents a few ways for entering the converted information into the ontology using the Protégé-OWL editor.

When Table 1 is completed the method continues with entering the converted information (or, in other words, the information from Table 1) into the ontology. This step can be divided in two. First, the classes for all defects that will be input in the ontology should be created as subclasses of class DefectID (as explained in Section 3.1.3). And second, the converted information about the defects (from Table 1) should be added to the created classes.

The editor we used for developing the ontology – Protégé-OWL, gives us three options for inputting the defects in the ontology. The first one is to manually create the classes for all defects and fill out all properties and the values of their respective ranges for every class. The second option is to use two editors – with the first one (called *Create Multiple Subclasses*) the classes are created, while with the second one (called *Quick Restriction Editor*) the information about the defects is added to the classes. The third option is to use the *Excel Import* plug-in – a batch importing plug-in from Protégé-OWL. It provides the opportunity to generate classes from the contents of Excel or CSV (comma-separated values) files. For example, the classes for the defects can be generated from the contents of the first column of Table 1. Then, to add the converted information to the created classes, restrictions are

generated or, in other words, the first column of Table 1 is related to the other columns via the properties defined in Section 3.1.2.

If Table 1 contains ten or twelve defects, for instance, then it will be relatively easy to enter them and their information manually into the ontology. If, however, the table contains many defects – twenty or more, then it is also possible to enter everything manually but it will be very time-consuming. Hence, in either case, it is more feasible not to use the first option. Completing the current step using the other two options is similar to a great extent. Since the second option provides important details for inputting the defects in the ontology, we used this option. The two editors are explained below together with some examples.

3.2.3.1. Creating the classes for the defects – editor *Create Multiple Subclasses*

Editor *Create Multiple Subclasses* is depicted in Fig. 13. The left side of the figure shows the steps that need to be completed in order to create the classes for all defects. This editor gives the opportunity to select the superclass that will contain all created classes (step 2 "Select superclass"). In this case, the superclass is class DefectID. Then, the names of the classes can be entered (or copied from other sources) and they will be created as subclasses of the selected superclass (step 3 "Enter names" – this step is shown in details in Fig. 13). It is valuable to know that prefixes and suffixes can be added for all class names and that the editor automatically validates that the entered terms are valid Protégé names. Moreover, the editor allows disjoints (step 4 "Disjointness") to be added automatically between all new siblings (and also between the new and the existing ones). We need to add disjoints since all defects are entered as separate and unique classes. In the end, the successful creation of the classes for all defects is confirmed by the editor (step 5 "Results").

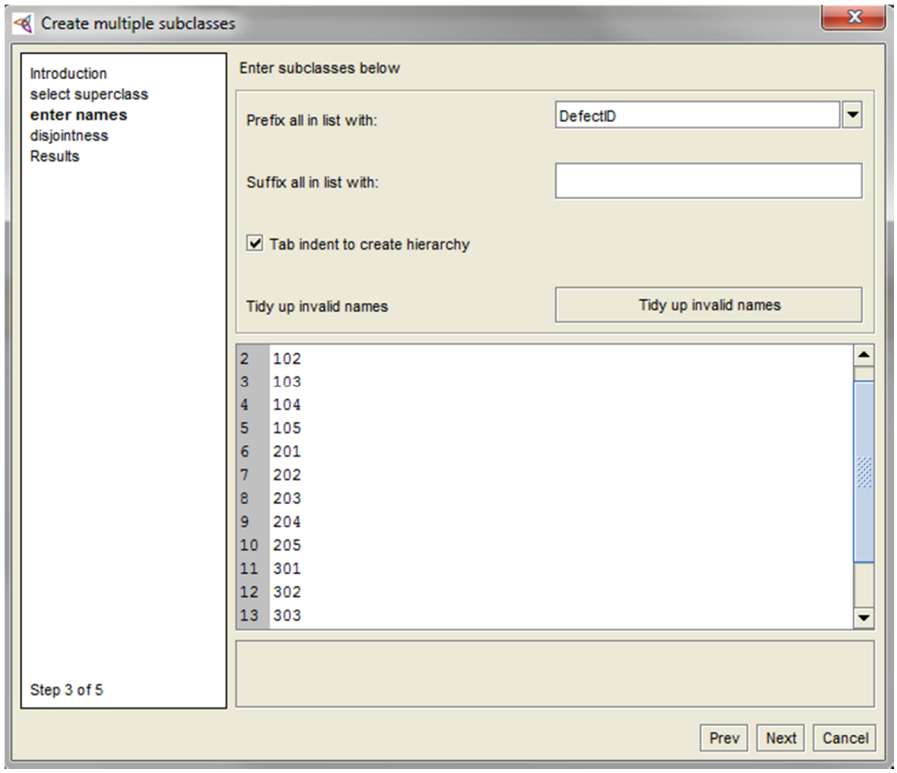

Figure 13. Editor *Create Multiple Subclasses*.

3.2.3.2. Adding the converted information to the classes – *Quick Restriction Editor*

Editor *Quick Restriction Editor* is depicted in Fig. 14. Similarly to above, the left side of the figure shows the steps that need to be completed in order to input the converted information in the ontology. The editor gives the opportunity to choose the classes that will be the domains (step 2 "Choose classes") and the object properties (step 3 "Choose properties") through multiple selections. Default values for the ranges of the chosen properties can be provided if needed (step 4 "Defaults"). Then, the values of the respective ranges can be edited through multiple selections or by choosing a value from a drop-down list (step 5 "Edit restrictions" – this step is shown in details in Fig. 14). An important and very useful feature of this editor is that it can create the closure axioms for all chosen object properties (step 6 "Closure"). As mentioned earlier, adding closure axioms for the chosen properties is essential since the axioms explicitly state that defects can have the values they are given and *only* these values. This way, the reasoner will be able to unambiguously classify the defects into the predefined severity levels and no inconsistencies will occur. In the end, the editor displays a message (step 7 "Results") confirming the successful creation of all relations (also called restrictions) between the chosen classes, object properties and values of the ranges.

This editor, however, does not allow using datatype properties and, therefore, their ranges have to be filled out manually for every defect input in the ontology.

In our research, we worked with more than thirty defects at all times. Thus, for creating the classes for all defects, the editor *Create Multiple Subclasses* was used. Then, for the four object properties, defined in Section 3.1.2, the *Quick Restriction Editor* was used to input the

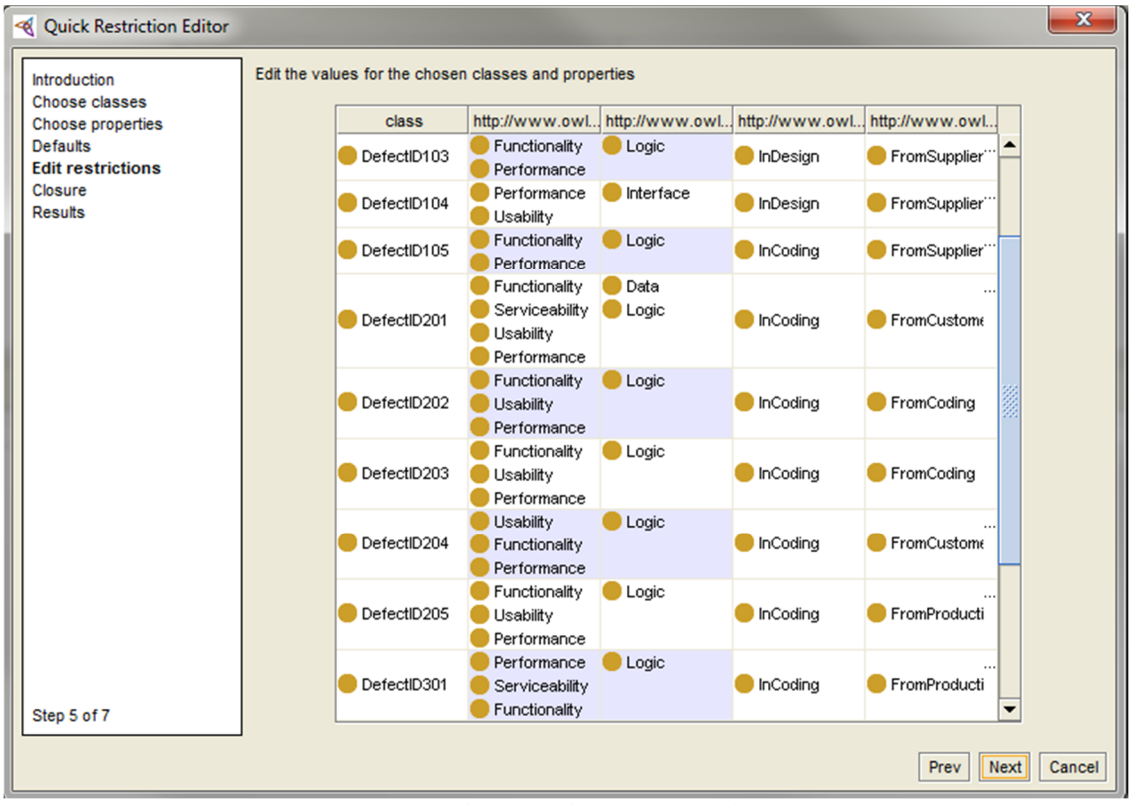

Figure 14. Editor *Quick Restriction Editor*.

values of the ranges for all defects and to create the respective closure axioms. After that, for the datatype property (Section 3.1.2), we input manually the values of its range for all defects.

In the end of this step, all defects and their converted information were input in the ontology. An example for the outcome from this step is illustrated in Fig. 15. It presents how the converted information about a defect looks like after the defect and its information are input in the ontology. This example features a defect with ID 205, which is represented by class DefectID205 in the ontology. As given on the left side of the figure, this defect is selected among the other defects in the ontology. The converted information about the defect is shown on the right side of the figure. Similar information is displayed for all defects input in the ontology upon selection.

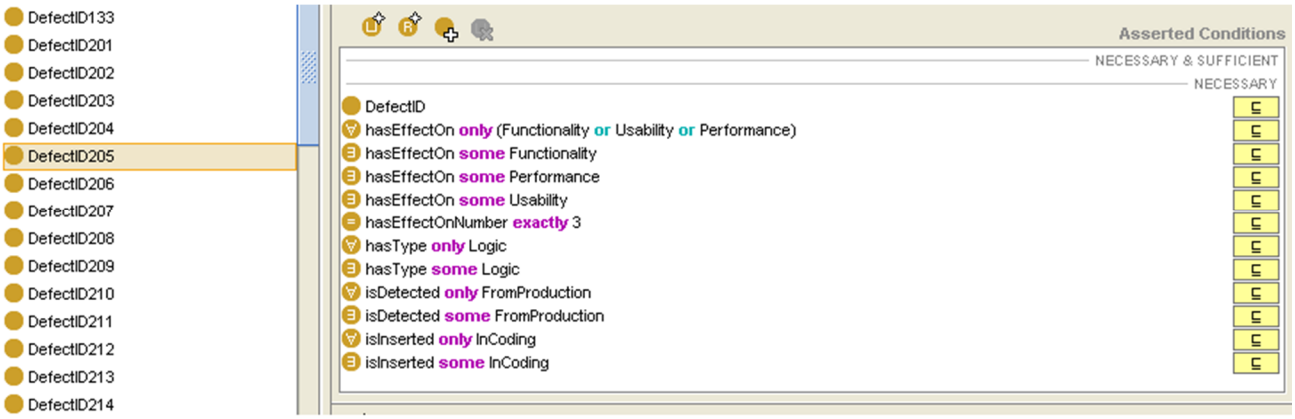

Figure 15. DefectID205 and its converted information after entering them into the ontology.

3.2.4. Automatically predicting the severity levels of defects

The last step in the method, once all defects and their information are input in the ontology, is to automatically predict the defects' severity levels. For achieving this goal, the input defects are automatically classified into the predefined severity levels using the developed rules (see Section 3.1.5) and the Pellet reasoner (see Section 2.5).

Defects are input in the ontology as separate classes, as explained in Section 3.2.3. Once this step is successfully completed, Pellet is employed to automatically classify the ontology hierarchy based on the developed rules. This is easily achieved by a single click of the "Classify taxonomy…" button available in the Protégé-OWL editor. A new window opens – depicted in Fig. 16, showing the process the reasoner goes through – synchronizing; checking the consistency of the ontology hierarchy; computing the inferred hierarchy and the equivalent classes (the reasoning phase). Upon the successful completion of this process, the inferred ontology hierarchy and the classification results are displayed in the editor. However, if the reasoner detects inconsistencies in the ontology hierarchy, it will display one or more classes, found to be inconsistent, in the window in Fig. 16. These classes and the respective defects should be checked. If the input information is incorrect or some of it is missing, it should be corrected and completed. Then, the classification of the ontology hierarchy should be repeated. The same applies if classes, which are not classified at all, are found while browsing through the inferred ontology hierarchy.

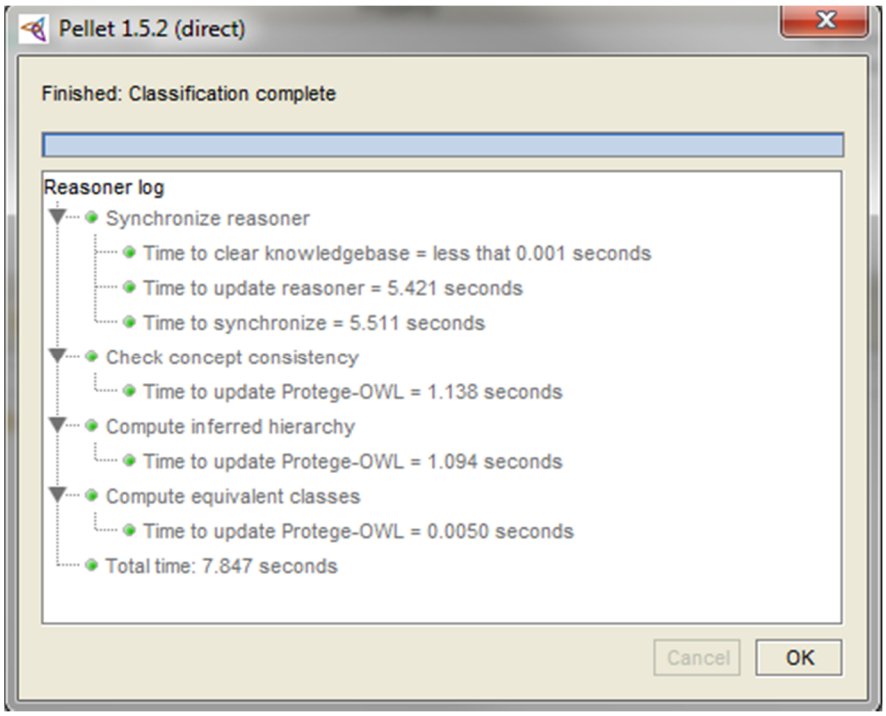

Figure 16. Pellet classification process.

The classification results are presented in Fig. 17. The left column contains the classes of every defect input in the ontology. The right column contains the classes that the classes from the left column are added to as subclasses. The classes in the right column represent the five severity levels from the IEEE standard. Hence, the classification results clearly show the severity level that is predicted for every defect input in the ontology. For example, in Fig. 17, class DefectID203 is added as a subclass to class DefectWithCriticalSL. Therefore, according to the developed method, this defect is predicted as having critical severity level.

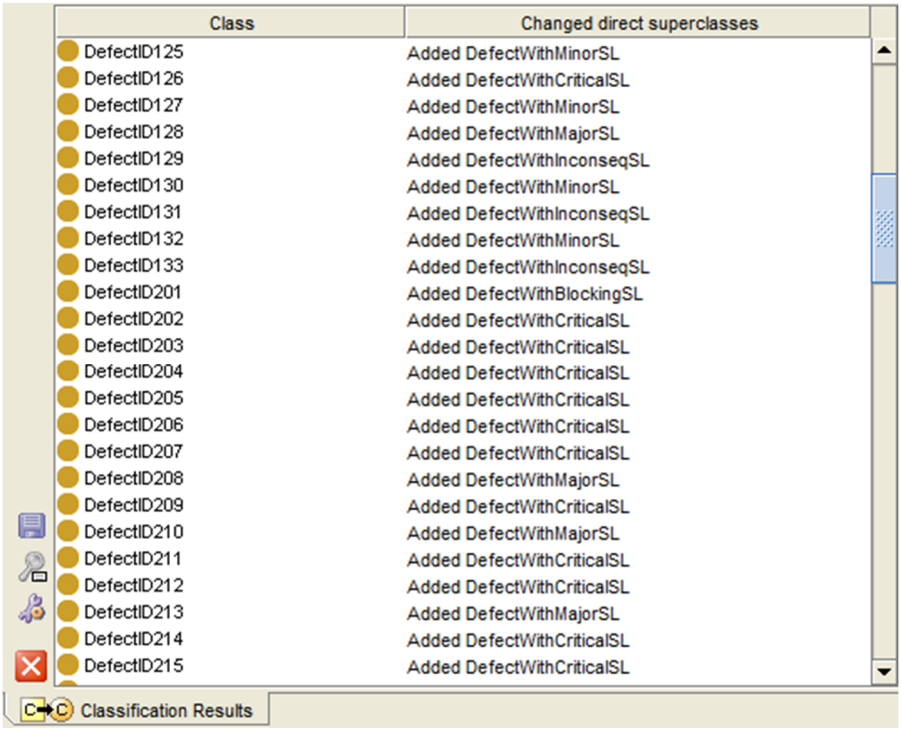

Figure 17. Classification results.

Figure 18 represents Fig. 11 after the ontology classification has been performed. Hence, for the five defects in Fig. 11, the reasoning phase has added links to the predicted severity levels. The layout of Fig. 11 and 18 is different from that of Fig. 6 (e.g., classes are denoted with ovals instead of rectangles) due to using a visualization tool that can display the results from the reasoning phase, as shown in Fig. 18.

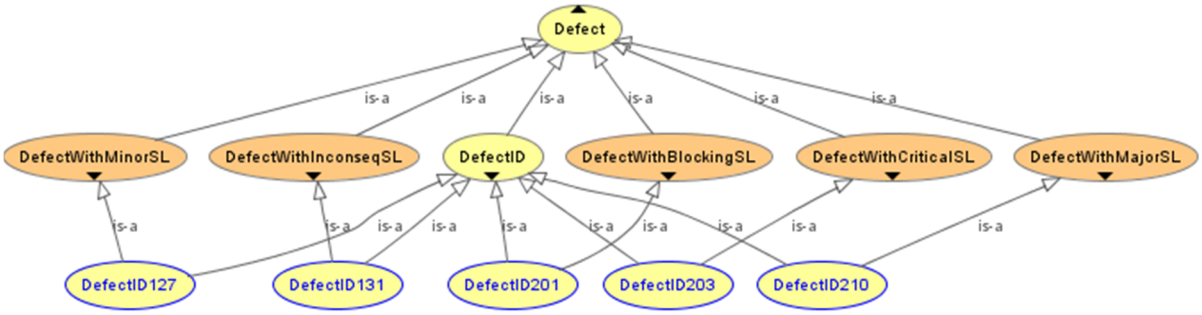

Figure 18. The five defects from Fig. 11 after performing the ontology classification.

In addition, the inferred ontology hierarchy, which is also a result from the classification process, can be used to easily browse through defects assigned a specific severity level. In other words, all defects, assigned blocking severity level, are grouped together as subclasses of class DefectWithBlockingSL. The same applies to the other defects and severity levels. This way, software teams can focus on fixing only defects assigned, for example, blocking- and critical severity level according to the period of time they have available.

3.3. Section summary

In this section we described the details of MAPDESO – the method for automated prediction of defect severity using ontologies. First, the process of developing the ontology was explained though this process is performed only once and it does not need to be repeated for using the method. Then, the details of the automated prediction method were presented in the same order in which the method could be applied to a real software project. The section ended with examples of results achieved by using the method.

The knowledge acquired until now is used to fully understand the approach applied to the case studies in Section 4, the validation in Section 5 and the comparison in Section 6.

4. CASE STUDIES

The two case studies in the research were conducted in an industrial environment – in the Technical Software Engineering Practice of Logica Netherlands B. V. (a company providing business consulting, technology and outsourcing services), in the company's office in Rotterdam, the Netherlands.

Both case studies follow the same approach. The approach is divided in three parts: data collection, data analysis and conversion, and data classification. As mentioned in Section 3, the classification rules were developed using the data from these two case studies. Thus, the data can be regarded as the training data for the developed ontology and rules.

4.1. Case Study 1

Reference [15] presents the approach and the results from Case Study 1 (CS1). However, since the completion of CS1, as explained in [15], we have improved a few things in the approach to the case study, which led to minor changes in the results. Hence, now we will present this improved version of the approach and the results from CS1.

Case Study 1 is based on a project for which Logica has developed the front-end software. The outcome from this project is an embedded traffic control system.

4.1.1. Data collection

The data represent fixed defects from the testing phase of the project. For keeping track of the defects found in the system during the testing phase, an issue management system (tracking system) has been used. Once a test analyst finds a defect, he inputs a corrective change request in the tracking system. It contains specific information about the defect such as its description, the version of the system that has been tested, where the defect originates from, the severity level of that defect (which is assigned by the test analyst), etc. The severity levels used in the project and as defined for the project are the following:

- Showstopper – a defect with such a severity level prevents the system from being put in production.
- Severe – a defect with such a severity level allows the system to be put in production if there is a workaround.
- Medium – a defect is assigned such a severity level if the system can work and this defect is not included in any other of the three categories of severity levels (this is the default severity level used in this project).
- Minor – a defect is assigned such a severity level if the system works but the users are annoyed by the defect.

The main part in the data collection step was to collect relevant and useful data for the purposes of the research. In order to do this, at least basic knowledge of the project in question was required. To gain such knowledge, the project was studied using its documentation – mainly design documents, UML diagrams, user manual, test documents.

These documents provided insights about the development of the project, its defect tracking system, the severity levels used in it and how to extract the required details about the defects. After that, the tracking system was used to extract a representative sample of 33 defects based on the project knowledge and the recommendations of the designers, the developers and the test analyst working on the project. This subset was selected to include defects from each severity level used in the project. The selected defects have been fixed in the latest version of the system (since only the latest version of the system contains the latest version of the design and the requirements), yet their number is limited because of time constraints. Table 2 presents details about the number of fixed defects according to the project's severity levels. The last column of the table shows the distribution of the selected defects according to the severities from the project.

Then, interviews were conducted with the same people working on the project to get detailed information about the selected defects and to verify that they are a representative subset (almost one third) of all defects fixed in the latest version of the system.

4.1.2. Data analysis and conversion

The detailed information about the 33 defects from the data collection step 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 [2] was used to convert the project-specific information about the defects into the project-independent attributes and their values defined in this standard. As explained in Section 1.3, the used attributes are the following: Severity, Effect, Type, Insertion activity and Detection activity. This conversion resulted in a table that contains the defect IDs (as used in the ontology) together with the values of the attributes from the standard assigned to each defect based on its detailed information. Table B.1.L in Appendix B presents the results from the conversion.

As already mentioned, the project used in CS1 has four severity levels. However, the ontology uses the severity levels from the IEEE standard (which provides five severity levels). Therefore, a relation should be defined that matches the severity levels from the project to the ones provided by the standard. This relation is defined in Table 3.

4.1.3. Data classification

The data classification step begins with entering the defects and the converted information about them from the previous subsection (see Table B.1.L in Appendix B) into the ontology.

Table 2. Number of fixed defects according to the severity levels from the project in CS1

Severity Level	*Number of Fixed Defects*		
	In all versions of the system	In the latest version of the system	Selected for Case Study 1
Showstopper	6	1	1
Severe	47	10	10
Medium	301	93	17
Minor	85	12	5
Total	439	116	33

Table 3. The relation between the severity levels for CS1

Severity Levels	
From the IEEE Standard Classification and used in the ontology	From the project used in Case Study 1
Blocking	Showstopper
Critical	Severe
Major	Medium
Minor	Minor
Inconsequential	Minor

Following the explanations in Section 3.2.3, we created the classes for the 33 defects as subclasses of class DefectID. These subclasses were named DefectID101, DefectID102, and so on up to DefectID133 because they are part of CS1 and their total number is 33. Next, the converted information about the defects was input in the ontology in the same way as explained in Section 3.2.3. It is visible from Table B.1.L that the input consists of the information about the defects concerning the values of attributes Effect (which represent quality properties), Type, Insertion activity and Detection activity.

As explained in Section 2.5, for the ontology development and the automatic classification we use the Protégé-OWL ontology editor with the OWL-DL language and the Pellet reasoner, respectively. Hence, the data classification step ends with the automatic classification of the defects into the predefined severity levels. Similarly to the process explained in Section 3.2.4, the Pellet reasoner classified all defects from CS1 input in the ontology into the five severity levels. The results from the classification were displayed in the ontology editor.

4.1.4. Results

The results can be found in Table B.1.R in Appendix B. The first column gives the original severity levels of the defects from the project while the second one contains the severity levels converted to the IEEE standard using the relation in Table 3. The third column presents the severity levels predicted by the developed method. The rows in Table B.1.R are distributed in such a way that they correspond to the rows in Table B.1.L for easy reference between the tables.

Next, we compared the results obtained using the automated prediction method with the results from the original classification (after applying the relation in Table 3). In other words, we compared the third column of Table B.1.R with the second column of the same table. Table 4 presents a summary of the results from the comparison between the two classifications using a confusion matrix. The numbers 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 classification. The remaining numbers (shown below the diagonal) represent the number of defects classified into higher severity levels by the ontology than by the original classification. Therefore, using the table, it can be easily calculated that the ontology classified 55% of the defects into the same severity levels

Table 4. Summary of the results from the comparison using a confusion matrix (CS1)

		Automatic (Ontology) Classification for CS1				
	Severity Levels	Blocking	Critical	Major	Minor	Inconse-quential
Manual (Original) Classifi-cation from CS1	Blocking	**1**	*0*	*0*	*0*	*0*
	Critical	*0*	**7**	3	*0*	*0*
	Major	*0*	7	**6**	2	2
	Minor	*0*	*0*	*0*	**2**	1
	Inconsequential	*0*	*0*	*0*	*0*	**2**

as originally. 24% of the defects were classified into lower severity levels by the ontology while 21% were classified into higher severity levels by the ontology than by the manual (original) classification. These results are summarized in Fig. 19.

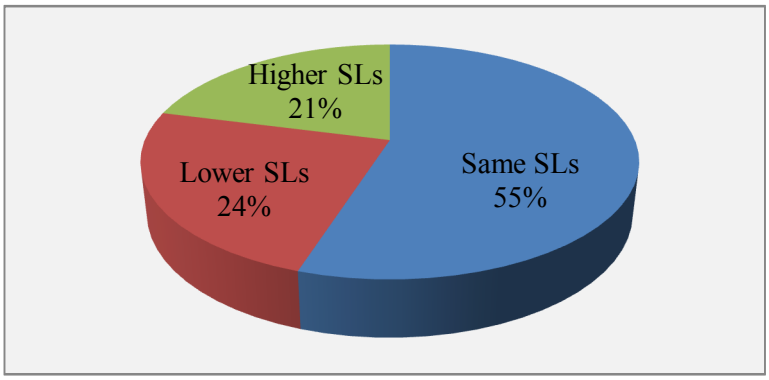

Figure 19. Percentages of the 33 defects (from CS1) classified into the same severity levels (SLs), lower SLs and higher SLs by the ontology compared with the original classification from CS1.

There are two reasons for the differences in the classification results. First, the ontology classification takes into account the point of view of the user of the software system while preserving the developer's point of view when considering which defects are important for fixing and which are not, as opposed to not taking into account the user's point of view at all (for more details see Section 3.1.5). 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 by the original classification (other factors also play a role in the classification process). 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 project without paying much attention whether this is the correct severity level or not. As mentioned in Section 4.1.1, the default severity level for the project is medium. However, using the relation in Table 3, we see that the default severity level is, in fact, major. So, Table B.1.L also contains such defects. Since the developed method classifies all defects, the defects assigned the default severity level in that table are assigned critical-, major-, minor- or inconsequential severity level by the ontology (as shown in Table 4). Hence, each defect is assigned a specific severity level and no default severity levels are used.

In the end, it should be taken into account that the results from CS1 were presented to two software architects from Logica familiar with the project used in CS1. Their opinion was the results satisfy the expectations that an automatic classification of defects into predefined severity levels is possible and the results from it are satisfactory for an initial case study.

Based on the achieved from CS1 and having in mind different projects use different sets of severity levels, we decided to work on another case study. Additional reason for doing so was to use data from a completely different project. The next subsection presents Case Study 2.

4.2. Case Study 2

Case Study 2 (CS2) is based on a project that Logica has been developing for eight years. Though the project is still in active development, it is already in use by the client. There are new releases regularly. The project is concerned with one main application with a couple of small utilities.

4.2.1. Data collection

In this case study the data represent fixed defects not only from the testing phase, as it was in CS1, but also from the post release use of the project. Although the project makes use of a different issue management system compared with CS1, it is easy to understand and use. When a defect is reported, which can be done by a developer, a test analyst or a user of the system, it is input in the defect tracking system. The additional information entered in the tracking system includes the defect's description, the release of the system where it has been detected, the origin of the defect, its severity level (which is assigned either by the test analyst or by the user or by both of them), etc. This project uses the following severity levels:

- Block – this severity level is assigned to defects that cause seriously reduced usability and, therefore, prevent the system from being released.
- Crash – a defect with such a severity level causes core dumps (it is not recommended to use the system in such cases).
- Major – this severity level is assigned to defects that cause logic problems leading to incorrect results (though the system can be used).
- Minor – this severity level is assigned to defects that cause cosmetic problems (this is the default severity level used in this project).

One of our concerns when collecting the data was to get relevant and useful data. Hence, we studied the project using its documentation – design documents and diagrams, user manuals and test documents. Once we gained some knowledge of the project, we continued with extracting the defects. The tracking system was used to select defects found during the testing phase and the post release use of the project. With the help of the software architect and the developers working on the project we extracted a sample of 47 defects. The reasons for selecting this subset were to include defects from each severity level from the project that have been fixed in the latest releases of the system (similarly to CS1), and to have more defects than in CS1 but still limit their number because we were constrained by time. Table 5 shows details about the number of fixed defects according to the project's severity levels. The last column presents the distribution of the selected defects according to these severities.

Table 5. Number of fixed defects according to the severity levels from the project in CS2

Severity Level	*Number of Fixed Defects*	
	In the latest releases of the system	Selected for Case Study 2
Block	1	1
Crash	11	11
Major	10	10
Minor	123	25
Total	145	47

After that, interviews were conducted with the software architect and a developer working on the project to get detailed information about the selected defects. We also verified that these defects are a representative subset (one third) of the fixed defects in the latest releases of the system.

4.2.2. Data analysis and conversion

The tracking system in this project provides very project-specific information about the defects. Hence, it is difficult to process the detailed information about the 47 defects from the previous step out of project context. To alleviate this, we used the IEEE standard in [2] and converted the project-specific information about the defects into the defect attributes and their values defined in the standard. The used attributes are the same as in CS1. The conversion resulted in Table B.2.L in Appendix B. Similarly to CS1, Table B.2.L contains the defect IDs (as used in the ontology) together with the values of the attributes from the standard assigned to each defect based on its information.

Moreover, there are four severity levels used in the project in CS2. However, the ontology uses five severity levels (from the IEEE standard). Therefore, we defined a relation that matches the severity levels from the project in CS2 to the ones in the ontology. Table 6 presents this relation.

4.2.3. Data classification

As in CS1, the data classification step begins with entering the defects and the converted information about them from the previous subsection (see Table B.2.L in Appendix B) into the ontology. Following the explanations in Section 3.2.3, we created classes for the 47 defects as subclasses of class DefectID. These subclasses were named DefectID201,

Table 6. The relation between the severity levels for CS2

Severity Levels	
From the IEEE Standard Classification and used in the ontology	From the project used in Case Study 2
Blocking	Block
Critical	Crash
Major	Major
Minor	Minor
Inconsequential	Minor

DefectID202, and so on up to DefectID247 because they are part of CS2 and their total number is 47. Next, the converted information about the defects was input in the ontology in the same way as explained in Section 3.2.3. Table B.2.L shows that the input consists of the information about the defects concerning the values of 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. Similarly to the process given in Section 3.2.4, the Pellet reasoner classified all defects from CS2 input in the ontology into the five severity levels. The end results from the classification were displayed in the editor.

4.2.4. Results

The results are shown in Table B.2.R in Appendix B. The first column contains the original severity levels of the defects from the project while the second one gives the severity levels converted to the IEEE standard using the relation in Table 6. The third column presents the severity levels predicted by the developed method. Similarly to the tables in CS1, the rows in Table B.2.R are distributed in such a way that they correspond to the rows in Table B.2.L for easy reference between the tables.

Next, we compared the results obtained using the automated prediction method with the results from the original classification (after applying the relation in Table 6). In other words, we compared the third column of Table B.2.R with the second column of the same table. Table 7 presents a summary of the results from the comparison between the two classifications using a confusion matrix. The numbers 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 classification. The remaining numbers (shown below the diagonal) represent the number of defects classified into higher severity levels by the ontology than by the original classification. As it can be seen on Fig. 20, the ontology classified 51% of the defects into the same severity levels as originally, 19% of the defects into lower severity levels and 30% into higher severity levels than originally.

The reasons for the differences in the classification results are very similar to those given in CS1. However, the default severity level in CS2 is minor (as opposed to major in CS1). Hence, defects that were originally assigned minor severity level are assigned major-, minor- or inconsequential severity level by the ontology (as shown in Table 7).

Table 7. Summary of the results from the comparison using a confusion matrix (CS2)

		Automatic (Ontology) Classification for CS2				
	Severity Levels	Blocking	Critical	Major	Minor	Inconse-quential
Manual (Original) Classifi-cation from CS2	Blocking	**1**	*0*	*0*	*0*	*0*
	Critical	*0*	**9**	2	*0*	*0*
	Major	*0*	4	**6**	*0*	*0*
	Minor	*0*	*0*	10	**7**	7
	Inconsequential	*0*	*0*	*0*	*0*	**1**

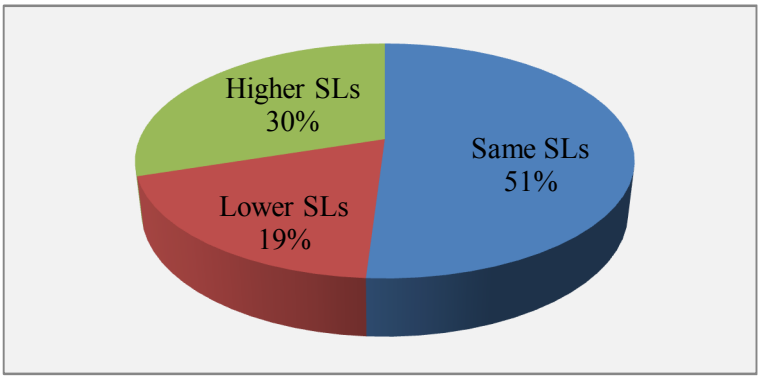

Figure 20. Percentages of the 47 defects (from CS2) classified into the same severity levels (SLs), lower SLs and higher SLs by the ontology compared with the original classification from CS2.

Last but not least, the results from CS2 were presented to the software architect and the team working on the project used in the case study. Their opinion was the results satisfy the expectations that an automatic classification of defects into predefined severity levels is possible and the results from it are quite satisfactory compared with the original classification results.

4.3. Summary of the results from Case Study 1 and Case Study 2

We gathered a total of 80 defects from both CS1 and CS2. In order to summarize the achieved results from both case studies we created the confusion matrix in Table 8. It compares the manual (original) classification of the defects from CS1 and CS2 with the automatic (ontology) classification of the same defects.

As before, the numbers 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 classification. The remaining numbers (shown below the diagonal) represent the number of defects classified into higher severity levels by the ontology than by the original classification. From Table 8, 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% of the defects into higher severity levels than the original classification. These results are summarized in Fig. 21.

The reasons for the differences in the classification results shown in Table 8 represent the addition of the reasons given in CS1 (Section 4.1.4) and those given in CS2 (Section 4.2.4).

Table 8. Summary of the results from the comparison using a confusion matrix (CS1 and CS2)

		Automatic (Ontology) Classification for CS1 & CS2				
	Severity Levels	Blocking	Critical	Major	Minor	Inconse-quential
Manual (Original) Classifi-cation from CS1 & 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
	Inconsequential	*0*	*0*	*0*	*0*	**3**

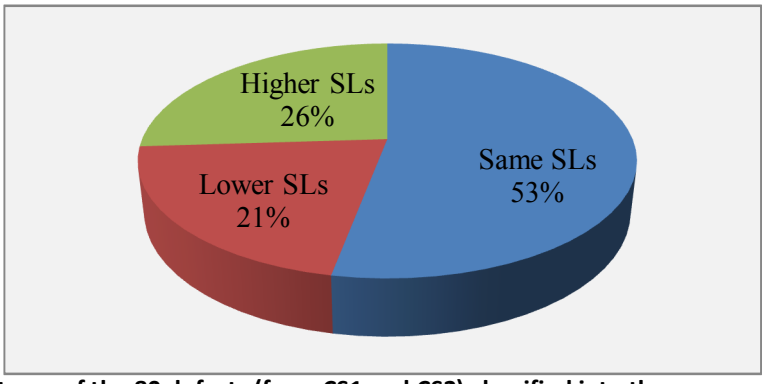

Figure 21. Percentages of the 80 defects (from CS1 and CS2) classified into the same severity levels (SLs), lower SLs and higher SLs by the ontology compared with the original classifications from CS1 and CS2.

4.4. Section summary

Summing up this section, it should be noted that since the gathered data from CS1 and CS2 can be considered as the training data for MAPDESO, the method has to be tested, too. Hence, the next step is to conduct a validation case study and the data from it will serve as the test data for MAPDESO. The successful completion of such a case study will validate the results achieved by the method when applied to an unknown project.

5. VALIDATION

In the beginning of this section, it should be emphasized that for the validation the already developed ontology and rules were tested on new data from a different project. Similarly to the two case studies from Section 4, the validation was also conducted in an industrial environment, namely at Logica. It consists of a Validation Case Study (VCS) and a small experiment at the end of the case study. VCS is based on a project whose development is completed. Currently, it is in production use and Logica provides its maintenance. The project represents an application that handles the messages between different companies and operators in order to make everything smooth for the clients. For working on VCS the steps below were taken.

5.1. Approach – VCS

The approach is divided in three: data collection, data analysis and conversion and data classification. An important difference of this approach compared with the one in Section 4 is that in VCS the severity levels of the selected defects were excluded from the defect reports. The details of the VCS approach are given below.

5.1.1. Data collection

The data represent fixed defects detected not only from the testing phase of the project but also during its maintenance. Similarly to the other two case studies, a defect can be reported by a developer, a test analyst or a user of the system. Once reported, the defect is input in the tracking system providing description for it, the date it was detected, the person who detected it, its severity level (assigned by a software team member or by a user, or by both), etc. A software engineer working on the project revealed that the project uses four severity levels:

- Top – assigned to defects that hinder the overall use of the system.
- High – assigned to defects causing disturbances in several components of the system.
- Medium – assigned to defects that cause disturbance(s) in a single component but do not prevent the use of the system.
- Low – assigned to defects causing minor inconveniences or cosmetic issues; the system works if such defects are present (this is the default severity in the project).

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 50 defects also because this number is just a bit more than the number of selected defects in each of the previous case studies. As before, this subset included defects from each and every severity level from the project. Table 9 presents the distribution of the 1163 defects

Table 9. Number of fixed defects according to the severity levels from the project in VCS

Severity Level	*Number of Fixed Defects*	
	In the received database	Selected for the validation of the method
Top	32	2
High	180	9
Medium	328	16
Low	623	23
Total	1163	50

and of the 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.

The decision for selecting 50 defects with the distribution shown in Table 9 was made without consulting the contents of the defect reports from the received database in order to keep our objectivity. After that, we had a couple of meetings with a software engineer from the project's team in order to introduce him to the details of our research. The reason for doing so was to get his help with the data conversion step.

5.1.2. Data analysis and conversion

According to the received database and as expected, the defect reports contained project-specific information. Therefore, we asked the same software engineer to convert the project-specific information about the defects into the attributes and their values defined in the IEEE standard. Once again, the reason for this action was to reduce any influence that we might have on the data conversion step. In order to complete this step we had a few more meetings with the software engineer for further clarifications of our research goals and the format of the information that we need to input in the ontology. The conversion resulted in Table B.3.L in Appendix B. The table contains the defect IDs, as used in the ontology, together with the values of attributes Effect, Type, Insertion activity and Detection activity, assigned to each defect based on its information.

In addition, as given in Section 5.1.1, the project uses four severity levels. As we already know, the ontology uses the five severity levels from the IEEE standard. Thus, we defined a relation that matches these two sets of severities. Table 10 presents the relation.

Table 10. The relation between the severity levels for VCS

Severity Levels	
From the IEEE Standard Classification 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 (in fact, not used in the project)

5.1.3. Data classification

The first part of the data classification step is to input the defects in the ontology. To do that, we created classes for the 50 defects as subclasses of DefectID. They were named DefectID301, DefectID302 and so on up to DefectID350 because they are part of VCS which is the third case study we worked on and their total number is 50. After that, the converted information about these defects, as given in Table B.3.L in Appendix B, was input in the ontology.

The final part of the data classification step is to automatically classify the defects into the predefined severity levels. As in Section 3.2.4, the Pellet reasoner automatically classified all defects input in the ontology into the five severity levels. The classification results were displayed in the ontology editor.

5.2. Results

The results from VCS are given in Table B.3.R in Appendix B. The table is organized in a similar way as in the training case studies from Section 4. In other words, the first column contains the original severity levels of the defects from the project while the second one gives the severity levels converted to the IEEE standard using the relation in Table 10. The third column presents the severity levels predicted by the developed method. Moreover, the rows in Table B.3.R are distributed in such a way that they correspond to the rows in Table B.3.L for easy reference between the tables.

Once the predicted severity levels of the selected defects were present, we compared them with the severity levels from the original classification after applying the relation in Table 10. Hence, we compared the third column of Table B.3.R with the second column of the same table. A summary of the results from the comparison between the two classifications is presented in Table 11 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 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 Fig. 22.

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. The first reason stems from the fact that there are defects assigned the default severity level in the project used for VCS. However, the default severity in it differs from the ones in the previous projects. As given in Section 5.1.1, this project's default severity level is low. Using Table 10, we see that it converts to minor. So, Table B.3.L also contains defects assigned the default severity. Since the ontology classifies all defects, the defects assigned the

Table 11. Summary of the results from the comparison using a confusion matrix (VCS)

		Automatic (Ontology) Classification for VCS				
	Severity Levels	Blocking	Critical	Major	Minor	Inconse-quential
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
	Inconsequential	*0*	*0*	*0*	*0*	**0**

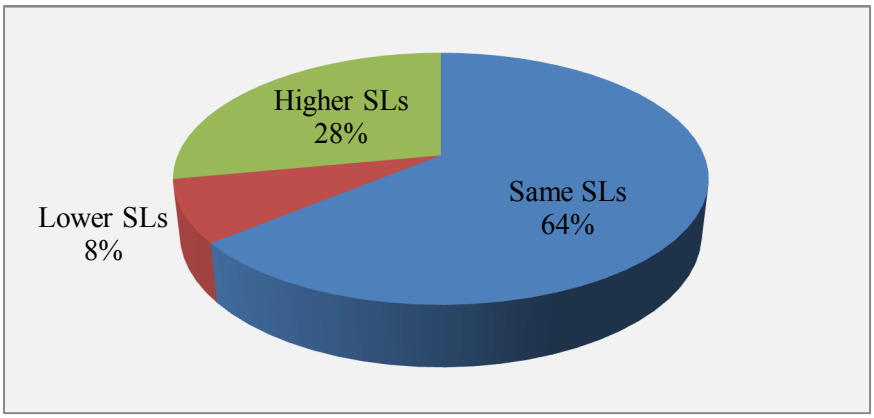

Figure 22. Percentages of the 50 defects (from VCS) classified into the same severity levels (SLs), lower SLs and higher SLs by the ontology compared with the original classification from VCS.

default severity level in that table are assigned critical-, major-, minor- or inconsequential severity level by the automatic classification (refer to Table 11).

Moreover, as pointed out in Section 3.1.5, the way the classification rules were developed implies that the method 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. This notion is illustrated with a couple of examples. First, the defect with ID 346 from Table B.3.L was originally assigned major severity level according to Table B.3.R. However, the automatic classification predicts for it critical severity level (see Table B.3.R). The combination of the values representing this defect implies that the users of the software will be affected more by this defect than originally anticipated when it was assigned major severity level. For this reason, the automatic classification predicts for it critical severity level. Therefore, this defect will be fixed sooner than initially planned and its fix will be included in the next release of the software. And second, the defects with IDs 307, 320 and 333, for example, are predicted as having the same severity levels as originally assigned. In this way, the automatic classification also aims at preserving the point of view of developers and test analysts when predicting which defects are important for fixing and which are not.

Before continuing further, we should point out that after comparing Fig. 22 with Fig. 21, we notice that the results from VCS are better than the results from the training cases. This can be contributed to the fact that we have dealt with very reliable defect and severity data. In addition, to confirm the above observation and, hence, the successful completion of VCS, we validated the results obtained from VCS. After that, we conducted a small experiment with

the help of the service coordinator for the project. The details of these two steps and the results from them are presented in the following two subsections.

5.3. Validation of the results

Validating the above results included presenting them to the software engineer and the service coordinator mentioned earlier. Then, by interviewing them, we found out their opinion about the performance of MAPDESO compared with the performance of the original (manual) classification. Since they played an important role in conducting VCS and they were aware of our research goals, it was straightforward to present and discuss the results with them. The main points that they have highlighted are mentioned below.

- The automated prediction method has performed surprisingly well compared with the original classification from the project – there are so many defects classified into the same severity levels by the ontology as originally especially having in mind that the method uses only four attributes from the IEEE standard to predict a fifth attribute – the severity levels.
- If only the defects classified differently (total number of mistakes) are considered, then their number should be low. However, as far as the majority of them are classified into higher severity levels by our method than by the manual classification (which is the case in Table 11), the automated prediction performs very well. In other words, it is good that there are more defects classified into high severity levels (critical, major) than into low severity levels (minor, inconsequential).
- It is very practical that the method uses an IEEE standard for the defects' attributes and their values. Although it might be difficult to use this method for current projects (because everybody is already using their predefined sets of severity levels) it should be applicable for future projects. Hence, future projects will have a standardized framework for the defects' attributes, which implies that people will be able to move from project to project, if needed, without wasting extra time for retraining.
- The method could be very useful for classifying many defects automatically and, then, focusing on the defects predicted as having severity level critical and above, for example. If necessary, the predicted severity levels could be checked manually and, after that, the defects would be fixed in the order of their severity levels.

The overall opinion of both the software engineer and the service coordinator was that MAPDESO yields very promising results. They also added that they would be very interested to see the method applied to other real projects.

5.4. Experiment

Before concluding the validation of the method, we conducted a short experiment with the help of the service coordinator for the project. The motivation for it stems from our willingness to explore the similarities and the differences between the originally assigned severity levels, the ones assigned by the service coordinator on the fly, and the ones predicted by the developed method. The experiment's setup and results are presented in this subsection.

For this experiment, we selected a representative sample of 20 defects out of the 50 defects initially selected for VCS. The distribution of the 20 defects according to the severity levels from the project is given in Table 12 together with the distribution of the 50 defects used for VCS. The table also shows these distributions according to the severity levels from the IEEE standard using the relation defined in Table 10. The reasons for this selection were the following: first, all parties involved in the experiment were constrained by time; second, the distribution of the 20 defects had to be the same as the distribution of the 50 defects (in terms of percentages) according to the severity levels (as evident in Table 12); lastly, the sample had to contain defects both on and off the diagonal in the confusion matrix (Table 11) instead of selecting defects only on the diagonal or only off it.

Once the selection was ready, the service coordinator used the available information about the 20 defects to assign severity levels to them. However, both the original and the predicted severity levels of these defects were removed prior to the experiment.

Next, we compared the service coordinator's classification of these 20 defects with the manual (original) classification from the project and with the automatic (ontology) classification. The comparison is presented in Table 13 using the severity levels from the IEEE standard (also used in the ontology). The defect IDs shown in the table correspond to the defect IDs shown in Table B.3.L in Appendix B. In order to easily recognize the similarities and the differences between the three classifications, Table 13 is constructed in a special way. The original classification from the project is given twice in the table (in the second and in the last columns) and the five severity levels are color-coded as **Blocking**, **Critical**, **Major**, **Minor** and **Inconsequential**.

The results from the comparison of each of the three classifications with the other two are summarized in three confusion matrices, three tables with percentages and three pie charts. First, Table 14 presents the confusion matrix summarizing the results from the comparison between the service coordinator's classification of the 20 defects and the original classification from the project. From the table we see that 65% of the defects (13 out of 20) were classified into the same severity levels by these two classifications. 15% of the defects were classified into lower and 20% into higher severity levels by the service coordinator than by the original classification. These percentages are summarized in Table 15 and visualized with a pie chart in Fig. 23.

Table 12. Distribution of the defects selected for the experiment according to the severity levels

Severity levels from the IEEE Std and used in the ontology	*Severity levels from the project used for the validation*	*Number of defects selected for the experiment*	*Number of defects selected for the validation*
Blocking	Top	1	2
Critical	High	4	9
Major	Medium	6	16
Minor	Low	9	23
Inconsequential	Low (not used)	0	0
Total defects:		20	50

Table 13. Comparison between the three classifications – all severity levels (SLs) are converted to IEEE Std

Defect ID	*Original SLs*	*SLs by the service coordinator*	*Predicted SLs*	*Original SLs*
303	**Major**	**Critical**	**Critical**	**Major**
304	**Blocking**	**Major**	**Blocking**	**Blocking**
305	**Major**	**Major**	**Minor**	**Major**
307	**Major**	**Major**	**Major**	**Major**
308	**Minor**	**Minor**	**Major**	**Minor**
309	**Minor**	**Minor**	**Minor**	**Minor**
310	**Minor**	**Minor**	**Critical**	**Minor**
314	**Major**	**Major**	**Major**	**Major**
315	**Critical**	**Critical**	**Critical**	**Critical**
322	**Minor**	**Minor**	**Minor**	**Minor**
323	**Critical**	**Minor**	**Major**	**Critical**
325	**Major**	**Minor**	**Major**	**Major**
329	**Minor**	**Minor**	**Minor**	**Minor**
336	**Minor**	**Critical**	**Major**	**Minor**
337	**Minor**	**Major**	**Minor**	**Minor**
341	**Minor**	**Minor**	**Inconsequential**	**Minor**
344	**Critical**	**Critical**	**Critical**	**Critical**
346	**Major**	**Major**	**Critical**	**Major**
347	**Minor**	**Minor**	**Minor**	**Minor**
348	**Critical**	**Blocking**	**Critical**	**Critical**

Table 14. Results from the comparison between the service coordinator's and the manual classifications

		Service Coordinator's Classification				
	Severity Levels	Blocking	Critical	Major	Minor	Inconse-quential
Manual (Original) Classifi-cation	Blocking	**0**	*0*	1	*0*	*0*
	Critical	1	**2**	*0*	1	*0*
	Major	*0*	1	**4**	1	*0*
	Minor	*0*	1	1	**7**	*0*
	Inconsequential	*0*	*0*	*0*	*0*	**0**

Table 15. Percentages of the 20 defects classified into the same severity levels (SLs), lower SLs and higher SLs by the service coordinator compared with the original classification from VCS

Comparison of	Percentages of the 20 defects classified into		
	Same SLs	Lower SLs	Higher SLs
The service coordinator's and the original classifications	65%	15%	20%

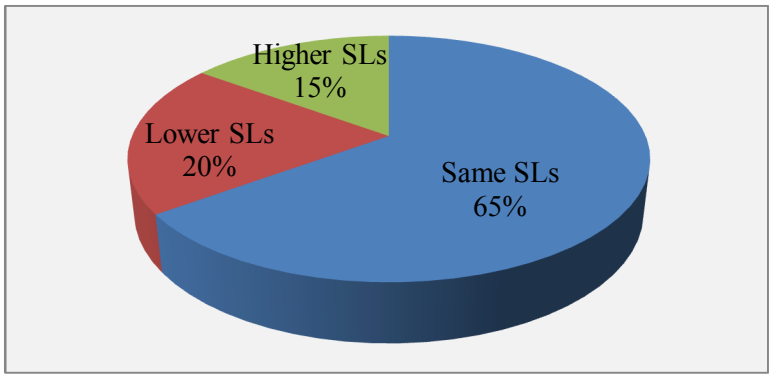

Figure 23. Percentages of the 20 defects classified into the same severity levels (SLs), lower SLs and higher SLs by the service coordinator compared with the original classification from VCS.

Table 16 presents the confusion matrix summarizing the results from the comparison between the service coordinator's classification and the automatic (ontology) classification. The service coordinator assigned to 45% of the defects the same severity levels as the ontology. 30% of the defects were classified into lower and 25% into higher severity levels by the service coordinator than by the automatic classification. These percentages are summarized in Table 17 and visualized with a pie chart in Fig. 24.

Table 16. Results from the comparison between the service coordinator's and the automatic classifications

		Service Coordinator's Classification				
	Severity Levels	Blocking	Critical	Major	Minor	Inconse-quential
Automatic (Ontology) Classifi-cation	Blocking	**0**	*0*	1	*0*	*0*
	Critical	1	**3**	1	1	*0*
	Major	*0*	1	**2**	3	*0*
	Minor	*0*	*0*	2	**4**	*0*
	Inconsequential	*0*	*0*	*0*	1	**0**

Table 17. Percentages of the 20 defects classified into the same severity levels (SLs), lower SLs and higher SLs by the service coordinator compared with the automatic (ontology) classification

Comparison of	Percentages of the 20 defects classified into		
	Same SLs	Lower SLs	Higher SLs
The service coordinator's and the automatic classifications	45%	30%	25%

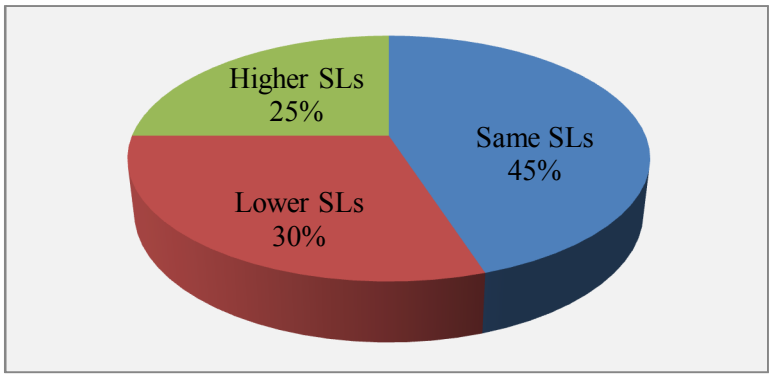

Figure 24. Percentages of the 20 defects classified into the same severity levels (SLs), lower SLs and higher SLs by the service coordinator compared with the automatic (ontology) classification.

Table 18 presents the confusion matrix that summarizes the results from the comparison between the automatic classification of the 20 defects and the original classification from the project. The ontology predicted 60% of the defects as having the same severity levels as originally, 15% of the defects as having lower severity levels than in the manual classification and 25% of the defects as having higher severity levels than in the manual classification. These percentages are summarized in Table 19 and visualized in the pie chart in Fig. 25. After that, Table 20 presents the contents of Tables 15, 17 and 19 combined together. Therefore, Table 20 shows the summary of the results from the comparisons in the experiment.

Table 18. Results from the comparison between the automatic and the manual classifications

		Automatic (Ontology) Classification				
	Severity Levels	Blocking	Critical	Major	Minor	Inconse-quential
Manual (Original) Classifi-cation	Blocking	**1**	*0*	*0*	*0*	*0*
	Critical	*0*	**3**	1	*0*	*0*
	Major	*0*	2	**3**	1	*0*
	Minor	*0*	1	2	**5**	1
	Inconsequential	*0*	*0*	*0*	*0*	**0**

Table 19. Percentages of the 20 defects classified into the same severity levels (SLs), lower SLs and higher SLs by the automatic classification compared with the original classification from VCS

Comparison of	Percentages of the 20 defects classified into		
	Same SLs	Lower SLs	Higher SLs
The automatic and the original classifications	60%	15%	25%

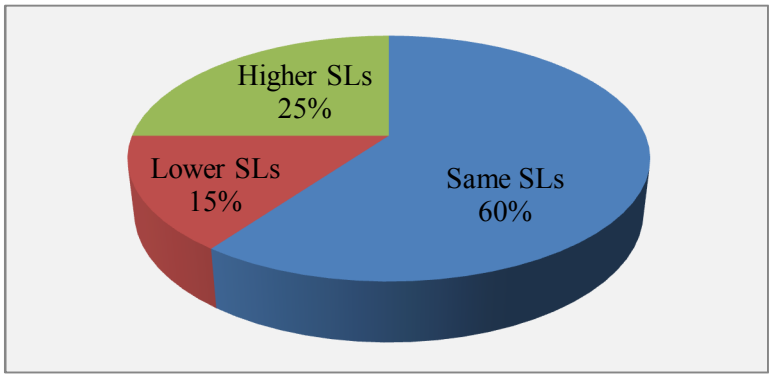

Figure 25. Percentages of the 20 defects classified into the same severity levels (SLs), lower SLs and higher SLs by the automatic classification compared with the original classification from VCS.

Table 20. Summary of the results from the comparisons in the experiment

Comparison of	Percentages of the 20 defects classified into		
	Same SLs	Lower SLs	Higher SLs
The service coordinator's and the original classifications	65%	15%	20%
The service coordinator's and the automatic classifications	45%	30%	25%
The automatic and the original classifications	60%	15%	25%

At the end, we presented the results from the experiment to the service coordinator and received his feedback on the similarities and the differences between the three classifications. The main reasons are summarized below.

- Sometimes defects are discussed with the client(s) and based on such discussions these defects are assigned lower severity levels (which are not necessarily the correct severity levels) than initially. The main reason for doing so is that if the severity level of a defect is lower, then the software team responsible for fixing it will have a bit more time to fix the defect. This fact plays a vital role in explaining why quite a few defects are assigned higher severity levels by the ontology classification than the other two classifications (as evident in Table 16 and in Table 18).
- The ontology classification considers quality properties affected by a defect (among other attributes) when assigning severity levels to defects. This leads to some defects assigned lower or higher severity levels by the ontology than by the other two classifications depending on what quality properties and other attributes are affected by the defects (see Table 16 and Table 18).
- In the original classification there are defects that are assigned the default severity level (minor). The service coordinator does not look at the severity levels at all. Hence, he is forced to assess every defect. This is a reason for the differences between the manual (original) classification and the service coordinator's classification (see Table 14). Moreover, as given in Section 5.2, this is also one of the reasons for the differences between the manual and the automatic classifications (see Table 18).

Using the service coordinator's opinion about the results from the experiment, we can conclude the following. If the predicted severity levels differ by at most one category compared with the original severity levels and the ones assigned by him, then the automated prediction method performs well. Table 13 shows that the only case not complying with this conclusion is the defect with ID 310. This defect is predicted as having critical severity level as opposed to having minor severity level according to the original classification (see Table 13). After looking into the details of the defect again, it was clear that, in addition to affecting three quality properties of the software, this defect directly affects the users. Hence, based on the ontology classification rules, it is predicted as having critical severity level in order to be fixed for the next release and satisfy the users of the software.

5.5. Section summary

This section presented the validation of MAPDESO. It was achieved through a case study and an experiment. Based on the results and on the opinion of the interviewees, it is safe to conclude that the automated prediction method yields promising results that can be very useful for medium-to-large projects with many defects.

In addition to the 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 from the comparison are presented in Section 6.

6. COMPARISON

This section presents the comparison of the performance of MAPDESO with the performances of algorithms from the Weka data mining software. However, in order 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 in Sections 3, 4 and 5, the data from CS1 and CS2 were used during the development of the ontology (as if they were training data) and the data from VCS were 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, while the data from VCS will be used for testing them. Moreover, in order to conclude the performance of which of the above two is better, the performance of each of them will be compared against the performance of the original (manual) classification from the project used for the validation of the method.

First, we will give some information about Weka. After that, we will present the process of predicting the severity levels of defects using classifiers from Weka. Then, we will present the comparison and the results from it together with the conclusion which of the above performs better and why.

6.1. The Weka machine learning workbench

The Weka[17] workbench is a collection of state-of-the-art machine learning algorithms and data preprocessing tools [19]. 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 [19].

Weka has been developed at the University of Waikato in New Zealand and the name stands for Waikato Environment for Knowledge Analysis [19]. The system is written in Java and it is distributed under the terms of the GNU General Public License. In short, Weka provides a uniform interface to different learning algorithms, together with methods for pre- and post-processing and for evaluating the results of learning schemes on any given dataset [19].

It should be mentioned that, as given in [19], there are three major ways of using Weka and they are the following:

- Apply a learning method to a dataset and analyze its output to learn more about the data.
- Use learned models to generate predictions on new instances.

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

- Apply several different learners and compare their performances in order to choose one for prediction.

Though Weka includes different tools and implementations, the most valuable resource that it provides are the implementations of actual learning schemes. We will use some of the learning methods for the comparison.

6.2. Predicting severity levels of defects using classifiers from Weka

The process of predicting the severity levels of defects using classifiers from Weka consists of a few steps. These steps include preparing the data to be used as input for Weka, selecting the classifiers (the learning algorithms), and classifying the test data using Weka. The second and third steps are completed using the Weka tool.

6.2.1. Preparing the data

Preparing the data to be used as input for a data mining algorithm usually consumes most of the effort invested in the entire data mining process [20]. Since the Weka package uses a specific file format, the input data for Weka has to be written in that file format. Weka uses the attribute-relation file format (ARFF format) which is a standard way of representing datasets that consist of independent, unordered instances and do not involve relationships among instances [20]. The ARFF format gives a dataset but it does not specify which is the attribute that is supposed to be predicted. In particular, this means that the same file can be used for investigating how well each attribute can be predicted from the others [20]. Hence, we can use this file format to predict the values of attribute Severity from the values of the other attributes, namely Effect, Type, Insertion activity and Detection activity.

We created two ARFF files containing all the data that we have in order to use them in Weka. The first one contains the data from CS1 and CS2 (a total of 80 defects), which will be used as the training data for the classifiers. In other words, the first ARFF file contains the data from Table B.1.L together with the second column of Table B.1.R and the data from Table B.2.L together with the second column of Table B.2.R. Of course, the original severity levels, as given in the second columns of Table B.1.R and Table B.2.R, have to be included in this ARFF file for training the classifiers.

The second ARFF file contains the data from VCS (a total of 50 defects), which will be used as the test data for the classifiers. Hence, the second ARFF file contains the data from Table B.3.L together with the second column of Table B.3.R. It should be noted that in this ARFF file we included the original (manually assigned) severity levels of the defects, as given in the second column of Table B.3.R. The reason for doing that is the following: once a classifier is applied to this ARFF file (representing the test dataset), Weka provides the results in terms of a confusion matrix and other statistics. This confusion matrix compares the severity levels predicted by the respective classifier with the original severity levels provided in the ARFF file. This information will be used later on for the comparison of the performances of the classifiers with the performance of the developed method.

6.2.2. Selecting the classifiers

After the data required for the input were in the correct file format, the next step was to choose the classifiers that will be used for predicting the severity levels of the defects.

As stated in [19], no single machine learning scheme is appropriate to all data mining problems. Therefore, in order to find out which classifiers will be appropriate for our task, we decided to compare the performances of fourteen classifiers available in Weka. As given in Section 6.1, one way of using Weka is to apply several different learners (classifiers) and compare their performances in order to choose one or more for prediction. We used Weka in this way in order to conclude which ones of the fourteen classifiers perform the best on the same dataset and to choose them for the prediction process and the comparison.

The fourteen classifiers, as they appear in the Weka workbench, are the following:

1) *ZeroR* – class for building and using a 0-R classifier.
2) *DecisionStump* – class for building and using a decision stump.
3) *NaiveBayes* – class for a Naive Bayes classifier using estimator classes.
4) *IB1* – 1-nearest-neighbor classifier.
5) *IBk* ($k = 5$) – 5-nearest-neighbors classifier.
6) *SimpleLogistic* – classifier for building linear logistic regression models.
7) *Logistic* – class for building and using a multinomial logistic regression model with a ridge estimator.
8) *LibSVM* – wrapper class for the LibSVM tools (LibSVM allows users to experiment with One-class SVM, Regressing SVM, and nu-SVM supported by LibSVM tool). SVM stands for Support Vector Machine.
9) *SMO* – classifier which implements John Platt's Sequential Minimal Optimization (SMO) algorithm for training a support vector classifier (a polynomial kernel was used for *SMO*).
10) *BFTree* – class for building a best-first decision tree classifier.
11) *DecisionTable* – class for building a simple decision table majority classifier.
12) *J48* – class for generating a pruned or unpruned C4.5 decision tree.
13) *RandomForest* – class for constructing a forest of random trees.
14) *RandomTree* – class for constructing a tree that considers K randomly chosen attributes at each node (performs no pruning).

For the comparison of these classifiers we used 10-fold cross-validation as method, the first ARFF file as dataset and area under the curve (AUC) as comparison field. The used method and comparison field are briefly described below.

It is clearly stated in [21] that 10-fold cross-validation yields the best estimate of error and that it has become the standard method in practical terms. This method works in the following way [21]: the data is divided randomly into 10 parts in which the class to be predicted is represented in approximately the same proportions as in the full dataset. After that, each part is held out in turn, the learning scheme (classifier) trained on the remaining nine-tenths and its error rate is calculated on the holdout set. Therefore, the learning procedure is executed a total

of 10 times on different training sets which have a lot in common. In the end, the 10 error estimates are averaged to yield an overall error estimate.

The Receiver Operating Characteristic (ROC) curves are a graphical technique for evaluating data mining schemes [21]. ROC curves depict the performance of a classifier without regard to class distribution or error costs. They plot the number of positives included in the sample, expressed as a percentage of the total number of positives, against the number of negatives included in the sample, expressed as a percentage of the total number of negatives [21]. Moreover, in order to summarize the ROC curves in a single quantity, often used is the area under the ROC curve, also called the Area Under the Curve (AUC). Since ROC curves plot numbers expressed as percentages, AUC represents a number between 0 and 1, including both. The area is interpreted as the probability that the classifier ranks a randomly chosen positive instance above a randomly chosen negative instance and, therefore, the larger the area the better the model [21]. In other words, if AUC is close to 0.5 then the classifier is practically random, whereas a number close to 1.0 means that the classifier makes practically perfect predictions [22].

The results from the comparison are summarized in Table 21. It contains all classifiers listed on the previous page together with the respective values of AUC, given in descending order. The first thing to notice is that *NaiveBayes*, *SMO* and *SimpleLogistic* perform the best out of the fourteen classifiers. This can be contributed to the fact that these three classifiers can work well when there is not a lot of data. In Section 6.2.1, we mentioned that 80 defects will be used as the training data and 50 defects will be used as the test data. Therefore, from a data mining perspective, this could be considered as not having a lot of data. We also see in the table that *DecisionStump* and *RandomForest* perform worse than the above three classifiers but still they have better than random performance. On the other hand, quite a few of the classifiers are practically random since their AUC values are 0.5 or close to 0.5. These are *BFTree*, *DecisionTable*, *ZeroR*, *IB1*, *LibSVM*, *RandomTree* and *J48*. It is not surprising to see

Table 21. Summary of the results from the comparison of fourteen classifiers

Classifier	Area Under the Curve (data from CS1 and CS2)
NaiveBayes	**0.81**
SMO	**0.74**
SimpleLogistic	**0.71**
DecisionStump	0.61
RandomForest	0.59
BFTree	0.52
DecisionTable	0.51
ZeroR	0.50
IB1	0.50
LibSVM	0.50
RandomTree	0.48
J48	0.43
IBk with $k = 5$	0.21
Logistic	0.14

that AUC for *ZeroR* is 0.5 because this classifier has AUC of 0.5 by definition. The last two classifiers are *IBk* ($k = 5$) and *Logistic* and they have the worst performances in the table. The performances of these nine classifiers can be explained with the fact that we did not have a lot of data for training, as we have already mentioned.

It is visible from Table 21 that the three classifiers that performed the best are *NaiveBayes*, *SimpleLogistic* and *SMO* (given in bold in the table). For this reason, we decided to use these three classifiers. However, we considered adding three more classifiers to the above three. We added *ZeroR*, *DecisionStump* and *IBk* with $k = 5$ (5-nearest-neighbors classifier) and they are underlined in the table. Although their performances are not as good as those of the first three classifiers, they are also well-known and widely-used. Moreover, we added them because of the following. First, *ZeroR* and *DecisionStump* are often used as worst-case reference classifiers and we wanted to have such classifiers for the comparison with MAPDESO. And second, we were interested to see how *IBk* ($k = 5$) will perform on the test dataset having in mind that it performed poorly in Table 21. More details about the six chosen classifiers are given below.

ZeroR is a simple classifier for generating rules. It predicts the test data's majority class (if nominal) or average value (if numeric) [23]. As we pointed out, *ZeroR* is typically used as a worst-case reference classifier.

DecisionStump builds one-level binary decision trees for datasets with a categorical or numeric class to be predicted. This classifier deals with missing values by treating them as a separate value and extending a third branch from the stump [23]. It is also typically used as a worst-case reference classifier.

NaiveBayes implements the standard probabilistic Naive Bayes classifier. It is based on Bayes's rule and "naively" assumes independence. It should be noted that the assumption attributes are independent in real life certainly is a simplistic one [24]. However, this method is easy to construct, not needing any complicated iterative parameter estimation schemes and it may be readily applied to huge datasets. Though it may not be the best possible classifier in any particular application, it can usually be relied on to be robust and to perform well [25].

IBk is a k-nearest-neighbor classifier, which finds the training instance closest in Euclidean distance to the given test instance and predicts the same class as this training instance [23]. In other words, this classifier finds a group of k objects in the training set that are closest to the test object and bases the assignment of a label on the predominance of a particular class in this neighborhood [25]. The number of nearest neighbors can be specified explicitly or determined automatically using cross-validation. As mentioned above, we use *IBk* with $k = 5$, which is a 5-nearest-neighbors classifier.

SimpleLogistic builds linear logistic regression models and determines how many iterations to perform using cross-validation, which supports automatic attribute selection [23]. More specifically, logistic regression builds a linear model based on a transformed target variable as opposed to linear regression, which attempts at approximating the target values directly [24].

SMO implements the sequential minimal optimization algorithm for training a support vector classifier using polynomial or Gaussian kernels (a polynomial kernel was used for *SMO*). When using this classifier, missing values are replaced globally, nominal attributes are transformed into binary ones, attributes are normalized by default and the coefficients in the output are based on the normalized data [23].

Once the classifiers were chosen, we continued with training them and classifying the data from the test dataset.

6.2.3. Classifying the test data

This step started with training the chosen classifiers using the Weka tool. As mentioned in Section 6.2.1, the data contained in the first ARFF file (the data from CS1 and CS2) were used as the training data for the classifiers. The training resulted in creating one model per chosen classifier based on the training dataset. Then, each of the created models was loaded in turn and the respective classifier was evaluated on the test data contained in the second ARFF file (the data from VCS). In the end, the results were displayed in Weka. In other words, once a classifier was trained on the first ARFF file, it "gained" knowledge, which was saved in a model. Then, the model was loaded and the classifier was evaluated on the test dataset. This process was repeated six times – once for each of the six chosen classifiers, and all results were displayed in the Weka tool.

The results provided by Weka include confusion matrices and various statistics. The confusion matrices compare the original (manually assigned) severity levels with the severity levels predicted by the classifiers. The statistics represent the detailed accuracies of the classifiers' predictions. They include true positive rate, false positive rate, precision, recall and F-measure for each of the severity levels. The weighted average values of each of these statistics are also calculated and displayed.

We have decided to use precision, recall and F-measure for the comparison of the performances in Section 6.3 in addition to the Percentages of Defects Classified Correctly (PDCC). These percentages represent the fractions of defects out of the defects in the test dataset that are predicted by the six classifiers as having the same severity levels as in the original classification (from the project used for VCS). In addition, the first three statistics are classification and information retrieval statistics. We chose these statistics (over AUC) because they provide useful and relevant to our research information for the performances of the classifiers, as evident by their definitions and mentioned in [21]. The statistics are defined below using the definitions provided in [21]. The definitions were adapted to the parameters in our research, namely defects and severity levels.

Precision represents the number of correct results divided by the number of all returned results. Therefore, precision can be seen as a measure of exactness or quality and it has its best value at 1 and worst value at 0. For our research, we have:

$$Precision = \frac{\text{number of defects that are correctly predicted as having a specific severity level}}{\text{total number of defects predicted as having that severity level}}$$

Recall is the number of correct results divided by the number of results that should have been returned. Hence, recall can be seen as a measure of completeness or quantity and, similarly to precision, it has its best value at 1 and worst value at 0. For our research, we have:

$$Recall = \\ = \frac{\text{number of defects that are correctly predicted as having a specific severity level}}{\text{total number of defects originally assigned that severity level}}$$

F-measure is a measure of a test's accuracy. It considers both the precision and the recall of the test to compute the score. The score can be interpreted as a weighted average of the precision and recall and the F-measure reaches its worst value at 0 and its best value at 1. It is calculated using the following formula:

$$F\ measure = \frac{2\ \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

Now that we have discussed the statistics we will use for the comparison, we present the results from classifying the test data using the six classifiers. Table 22 (on the next page) 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. Since all results were displayed in the Weka tool, as mentioned earlier, the results shown in Table 22 were directly taken from Weka.

6.3. Comparison of the performances

The comparison of the performances starts with the task of presenting the results from testing the developed method using the same four statistics we have decided to use. As mentioned in the beginning of this section, the performance of MAPDESO was tested using the data from VCS during the validation process (see Section 5). The results from the testing are given at the bottom of Table 22. We have calculated the percentage of defects classified correctly plus the precision, recall and F-measure per severity level together with the weighted average values of these statistics for the method.

The results from Table 22 are visualized in three figures. Figures 26, 27 and 28 present graphically the results for precision, recall and F-measure, respectively, per severity level for the six classifiers and for MAPDESO. Since the percentages of defects classified correctly represent the respective recall values multiplied by 100 (to convert them to percentages), there is no figure created for the PDCC results from Table 22 so as to avoid redundancy.

Table 22. The results from classifying the test data (VCS data) by the six chosen classifiers and by MAPDESO

Classifiers	*Severity levels*	*PDCC*	*Precision*	*Recall*	*F-measure*
Classifier 1: *ZeroR*	Blocking	0%	0	0	0
	Critical	0%	0	0	0
	Major	100%	0.32	1	0.49
	Minor	0%	0	0	0
	Inconsequential	0%	0	0	0
	W. Avg.	32%	0.10	0.32	0.16
Classifier 2: *DecisionStump*	Blocking	0%	0	0	0
	Critical	56%	0.25	0.56	0.35
	Major	0%	0	0	0
	Minor	83%	0.63	0.83	0.72
	Inconsequential	0%	0	0	0
	W. Avg.	48%	0.34	0.48	0.39
Classifier 3: *NaiveBayes*	Blocking	0%	0	0	0
	Critical	44%	0.36	0.44	0.40
	Major	50%	0.40	0.50	0.44
	Minor	57%	0.68	0.57	0.62
	Inconsequential	0%	0	0	0
	W. Avg.	50%	0.51	0.50	0.50
Classifier 4: *IBk* with $k = 5$	Blocking	0%	0	0	0
	Critical	56%	0.56	0.56	0.56
	Major	38%	0.50	0.38	0.43
	Minor	78%	0.62	0.78	0.69
	Inconsequential	0%	0	0	0
	W. Avg.	58%	0.55	0.58	0.56
Classifier 5: *SimpleLogistic*	Blocking	0%	0	0	0
	Critical	89%	0.35	0.89	0.50
	Major	13%	1	0.13	0.22
	Minor	83%	0.76	0.83	0.79
	Inconsequential	0%	0	0	0
	W. Avg.	58%	0.73	0.58	0.52
Classifier 6: *SMO*	Blocking	50%	1	0.50	0.67
	Critical	56%	0.46	0.56	0.50
	Major	6%	0.50	0.06	0.11
	Minor	96%	0.61	0.96	0.75
	Inconsequential	0%	0	0	0
	W. Avg.	58%	0.56	0.58	0.50
MAPDESO – automated prediction method	Blocking	100%	1	1	1
	Critical	89%	0.57	0.89	0.70
	Major	56%	0.50	0.56	0.53
	Minor	57%	0.87	0.57	0.68
	Inconsequential	0%	0	0	0
	W. Avg.	64%	0.70	0.64	0.65

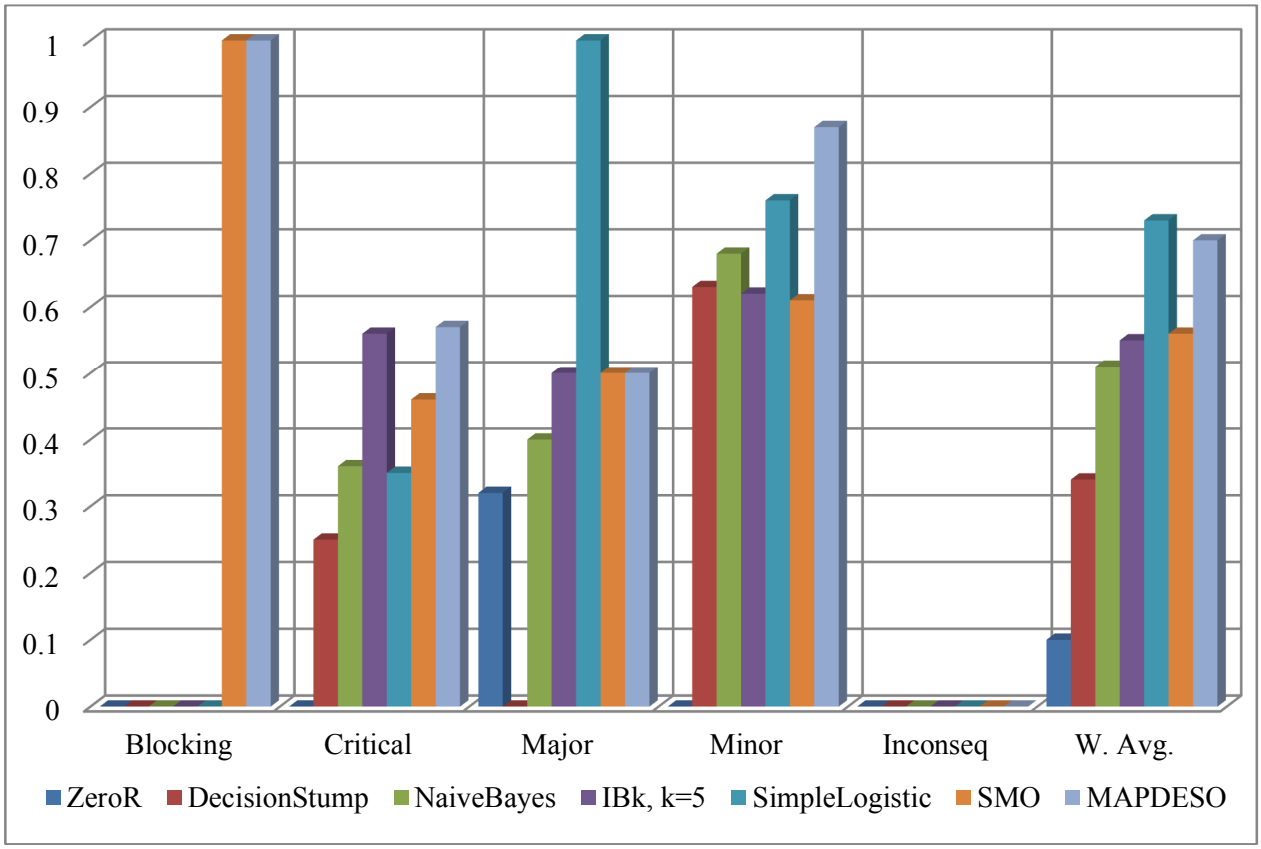

Figure 26. The results for precision per severity level for the six classifiers and for MAPDESO.

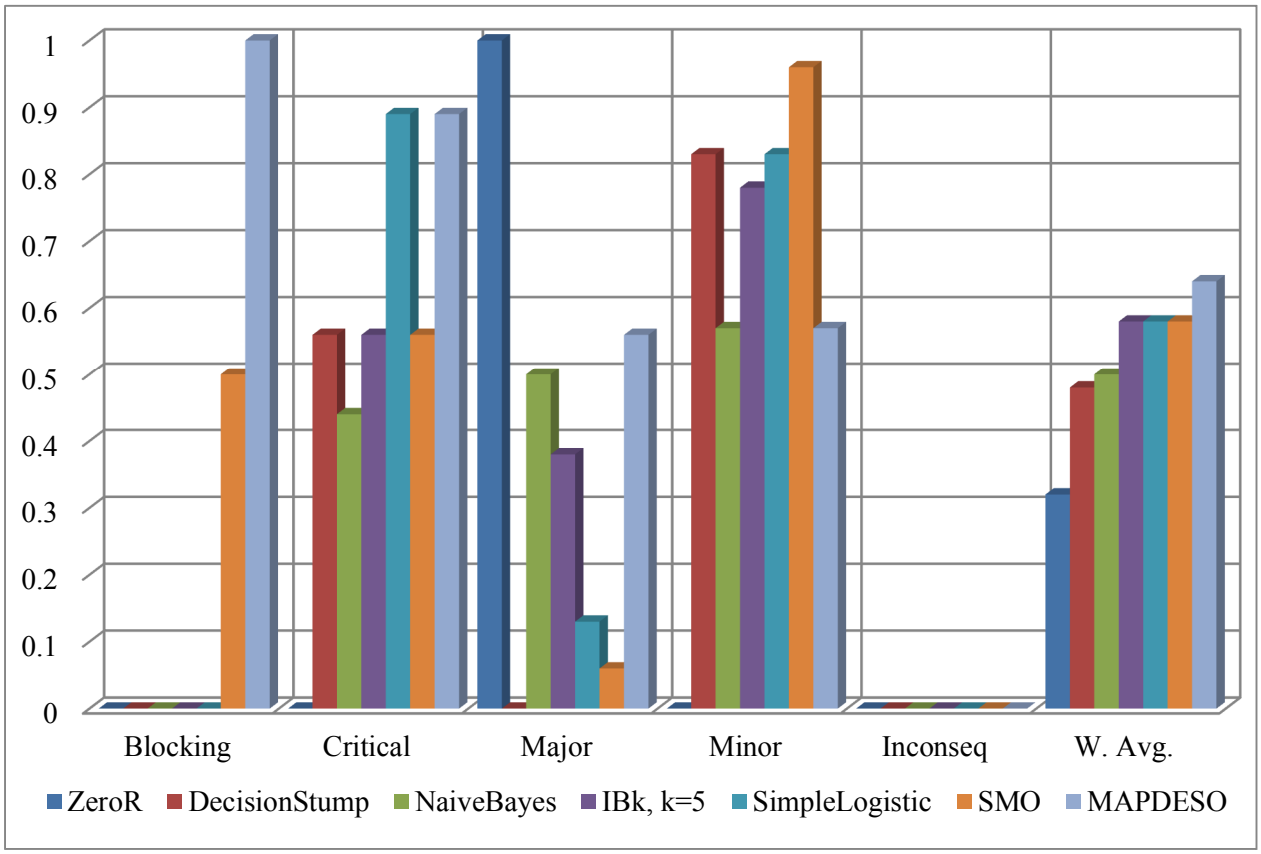

Figure 27. The results for recall per severity level for the six classifiers and for MAPDESO.

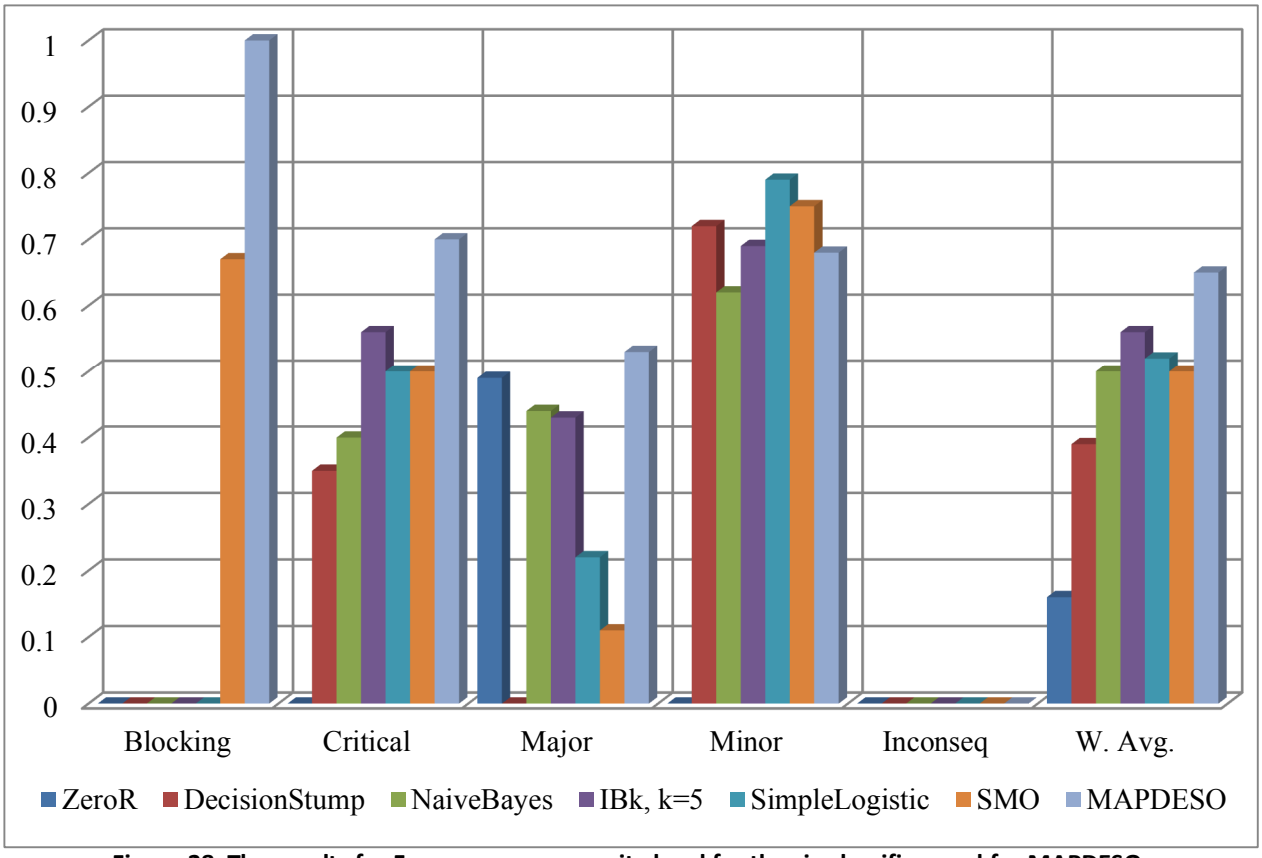

Figure 28. The results for F-measure per severity level for the six classifiers and for MAPDESO.

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 the four statistics. In addition to Fig. 26, 27 and 28, we created Table 23. It contains the weighted average values of the four statistics (taken from Table 22) for the classifiers and for the method. Then, the weighted average values of precision and recall were plotted together in a single chart, visualized in Fig. 29. As explained earlier, precision and recall reach their best values at 1. For Fig. 29, this means that these two statistics reach their best values in the upper right corner of the figure. Therefore, the closer a classifier is to the upper right corner of the figure, the better its performance will be. We see in the figure that MAPDESO and *SimpleLogistic* are the two closest to the upper right corner.

Table 23. Summary of the comparison between the six classifiers and MAPDESO

Classifiers (classification methods)	*Weighted average values of*			
	PDCC	*Precision*	*Recall*	*F-measure*
ZeroR	32%	0.10	0.32	0.16
DecisionStump	48%	0.34	0.48	0.39
NaiveBayes	50%	0.51	0.50	0.50
IBk with $k = 5$	58%	0.55	0.58	0.56
SimpleLogistic	58%	**0.73**	0.58	0.52
SMO	58%	0.56	0.58	0.50
MAPDESO – automated prediction method	**64%**	0.70	**0.64**	**0.65**

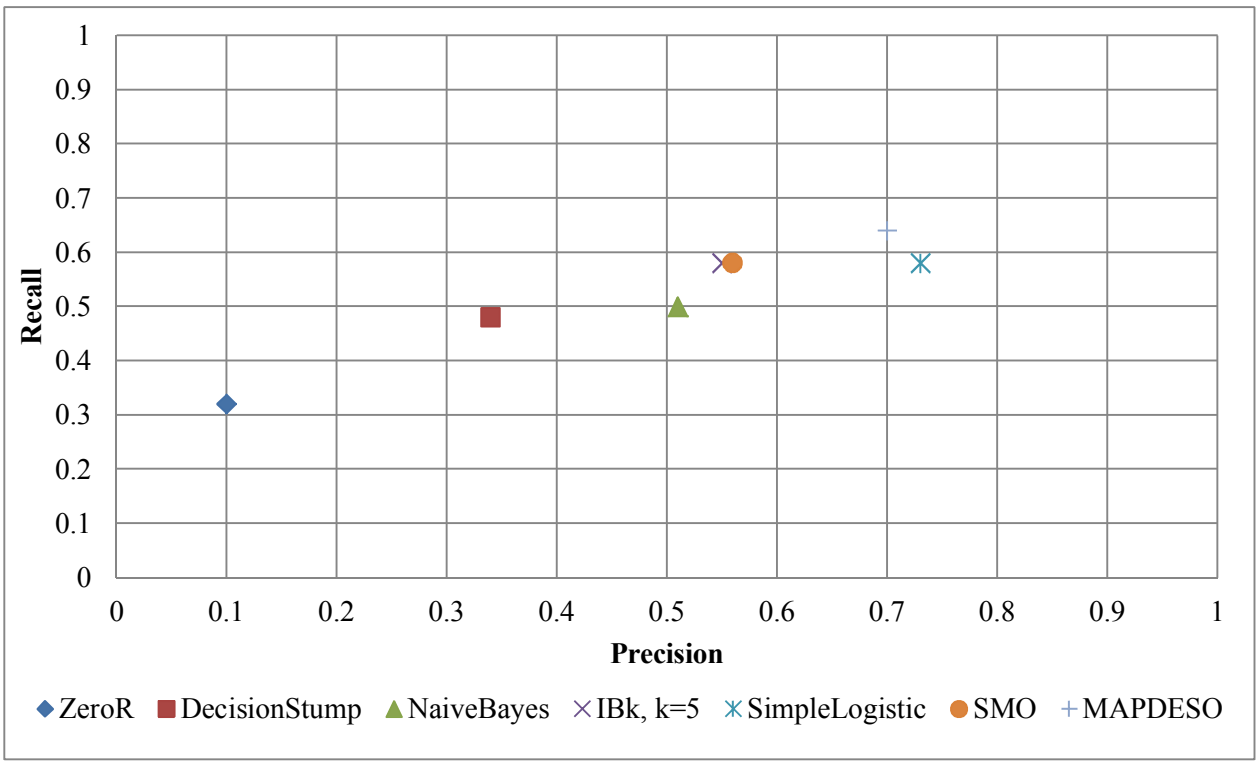

Figure 29. Weighted average values of precision and recall for the six classifiers and for MAPDESO.

It is visible that our automated prediction method has the highest percentage of defects classified correctly (Table 23), the second highest precision (Fig. 26, Table 23, Fig. 29), the highest recall (Fig. 27, Table 23, Fig. 29) and the highest F-measure (Fig. 28, Table 23). Only the *SimpleLogistic* classifier has a precision greater than that of the developed method (Fig. 26, Table 23, Fig. 29) and the reasons for this are explained below.

First, we have to look at the specific precision values for the different severity levels for the *SimpleLogistic* classifier (see Table 22 and Fig. 26). It is easy to notice that the precision for *SimpleLogistic* is 1 for severity level *major*. 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. From these observations we can conclude that although this classifier returns only correct results for severity level *major* (the precision is 1), it returns a very small portion of the correct results that should have been returned for this severity level (the 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 for this severity level (the recall is 0.56). So, the returned results are exact one half of the time and complete more than half of the time.

Moreover, if we look at the weighted average F-measure for this classifier, 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 Table 23) despite the fact that *SimpleLogistic* has a precision greater than

that of the method. Therefore, based on the above explanations and the results in Table 22, 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. In Fig. 26 and 27, we can see that for a specific severity level one or more classifiers might have a precision and/or a recall greater than or equal to those of the automated prediction method but for the other severity levels the method has greater values of precision and recall. Hence, based on Tables 22 and 23, on Fig. 26, 27, 28 and 29, and on everything explained in the current subsection, 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 Table 22 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. This means that the method performs the best compared with the performances of the classifiers when predicting which defects will be assigned the most important severities, namely *blocking* and *critical*.

6.4. Section summary

Summing up this section, we presented the comparison of the performance of MAPDESO with the performances of six classifiers chosen from the Weka machine learning workbench. The Weka software was used for completing the required steps for the comparison. Based on the achieved results, we concluded that the overall performance of the automated prediction method is better than the performances of the Weka classifiers and it reaches its peak when predicting which defects will be assigned the most important severity levels – *blocking* and *critical*.

7. CONCLUSIONS AND RECOMMENDATIONS

In this thesis, 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 [2] to create a uniform framework for reporting the defects and to make 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 thesis 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 and a small experiment. In the end, the method's performance was compared with 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 has performed surprisingly well compared with the manual (original) classifications of the defects from the conducted case studies especially having in mind that 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 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.
- ✓ The automated prediction method 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*.

These are very exciting results since they speak to the usability of MAPDESO. A few recommendations to use the method in practice are outlined below.

One way to apply MAPDESO in practice is to implement it as an addition to an existing defect tracking system. If one inputs the required information about the defects and chooses to use the method, he/she will get the severity levels predicted automatically. Hence, the default severities, assigned initially, will be improved and such a situation could lead to many people using the method regularly.

Another option is to implement a new tracking system that makes use of the ontology for keeping track of the defects. The defects will be entered into the tracking system by inputting them in the ontology using the attributes from the IEEE standard. If entering additional information about the defects is required by a project, the ontology provides a description field for it. Once the defects are in the tracking system, the prediction of their severity levels is done automatically.

One other way of applying MAPDESO in practice is to use the method indirectly. In other words, once the severity levels are automatically predicted, they could be considered only as suggestions. These suggestions could be used in two ways: (1) before software engineers and/or clients assign severity levels to defects so that they can consult the suggested severities; or (2) after severity levels are manually assigned to defects so that engineers/clients will be alerted to the differences between the classifications in order to consider any changes. Once the suggested severity levels are confirmed, they could be used for prioritizing the fixing activities of the defects.

Currently, MAPDESO predicts the severity levels of defects detected from system-level testing, coding and maintenance. However, MAPDESO could be easily tuned so that it can be used to predict the severity levels of defects detected from any phase of the software development process.

Last but not least, parts of MAPDESO could be used for defining service-level agreements (SLA). SLA is a part of a service contract (between a provider and a consumer) where the level of service is formally defined. For instance, software companies and their clients could agree to use the IEEE standard for reporting the defects in software projects. If this is achieved, the method could be directly applied to the defect reports and it will predict automatically the severity levels of the defects.

These possibilities to use MAPDESO in practice could also be considered as future work. More information about our future work is given in the next section.

8. 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.

As mentioned in the previous section, future work would also be aimed at applying MAPDESO in practice. This could be achieved by implementing it in a defect tracking system either as the sole method for predicting the severity levels of defects or as a method providing severity levels as suggestions that will be confirmed by software engineers and/or clients.

A possible continuation of this work is to apply the automated prediction method to other projects, for example open-source projects. Ideally, people involved in such a project will be available for discussions and interviews in order to: (1) extract defects from the tracking system of the project; (2) convert the extracted information into the defect attributes and their values used in the ontology; (3) validate the severity levels predicted by the method once it is ready. Similarly to Section 5, in the end, we would like to conclude how well MAPDESO performs compared with the original classification from the used project.

Last but not least, we would also like to try blending machine learning with our method. In an ideal world, we could use the combination of data and knowledge in order to get the best of both. For example, data could be used to infer relationships based on the available evidence, while knowledge could be used when the data is not abundant but theory is available and known to be stable.

BIBLIOGRAPHY

[1] *IEEE Standard Computer Dictionary. A Compilation of IEEE Standard Computer Glossaries*, IEEE Std 610-1991, doi:10.1109/IEEESTD.1991.106963.

[2] *IEEE Standard Classification for Software Anomalies*, IEEE Std 1044-2009, doi:10.1109/IEEESTD.2010.5399061.

[3] T. Menzies and A. Marcus, "Automated Severity Assessment of Software Defect Reports," *Proc. IEEE Int. Conf. on Software Maintenance*, Beijing, 2008, pp. 346-355, doi:10.1109/ICSM.2008.4658083.

[4] Y. Zhou and H. Leung, "Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults," *IEEE Trans. Softw. Eng.*, vol. 32, no. 10, Oct. 2006, pp. 771-789, doi:10.1109/TSE.2006.102.

[5] M. Termeer. (2005, Jan. 16). *MetricView* [Online]. Available: http://www.win.tue.nl/empanada/metricview/.

[6] M. D. B. M. Suffian, "Defect Prediction Model for Testing Phase," M.S. thesis, Faculty Comput. Sci. Inform. Syst., Univ. Teknologi Malaysia, Johor, Malaysia, 2009.

[7] A. M. Hoss, "Ontology-Based Methodology for Error Detection in Software Design," Ph.D. dissertation, Dept. Comput. Sci., Louisiana State Univ., Baton Rouge, 2006.

[8] Y. Kalfoglou, "Deploying Ontologies in Software Design," Ph.D. dissertation, Dept. Artificial Intell., Univ. Edinburgh, Edinburgh, UK, 2000.

[9] D. Jin and J. R. Cordy, "Ontology-Based Software Analysis and Reengineering Tool Integration: The OASIS Service-Sharing Methodology," *Proc. 21st IEEE Int. Conf. on Software Maintenance*, Budapest, 2005, pp. 613-616, doi:10.1109/ICSM.2005.68.

[10] T. R. Gruber, "A Translation Approach to Portable Ontology Specifications," *Knowledge Acquisition*, vol. 5, no. 2, Jun. 1993, pp. 199-220, doi:10.1006/knac.1993.1008.

[11] N. F. Noy and D. L. McGuinness, "Ontology Development 101: A Guide to Creating Your First Ontology," Stanford Knowledge Syst. Lab. Tech. Rep. KSL-01-05 and Stanford Medical Informatics Tech. Rep. SMI-2001-0880, Stanford Univ., Stanford, CA, Mar. 2001.

[12] L. Dittmann, T. Rademacher, and S. Zelewski, "Performing FMEA using ontologies," *18th Int. Workshop on Qualitative Reasoning*, Northwestern Univ., Evanston, IL, Aug. 2-4, 2004.

[13] M. Horridge, H. Knublauch, A. Rector, R. Stevens, and C. Wroe, "A Practical Guide to Building OWL Ontologies Using the Protégé-OWL Plugin and CO-ODE Tools Edition 1.0," Univ. Manchester, Manchester, UK, Aug. 27, 2004.

[14] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz, "Pellet: A Practical OWL-DL Reasoner," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 5, no. 2, Jun. 2007, pp. 51-53, doi:10.1016/j.websem.2007.03.004.

[15] M. Iliev, B. Karasneh, M. R. V. Chaudron, and E. Essenius, "Automated Prediction of Defect Severity Based on Codifying Design Knowledge Using Ontologies," *Proc. 2012 1st Int. Workshop on Realizing Artificial Intell. Synergies in Software Eng. (RAISE)*, Zurich, Jun. 5, 2012, pp. 7-11, doi:10.1109/RAISE.2012.6227962.

[16] D. Allemang and J. Hendler, "What is the Semantic Web?" in *Semantic Web for the Working Ontologist: Effective Modeling in RDFS and OWL*, 2nd ed., Waltham, MA: Morgan Kaufmann Publishers, 2011, ch. 1, pp. 1-2, doi:10.1016/B978-0-12-385965-5.10001-9.

[17] J. Cardoso, "The Semantic Web Vision: Where are We?", *IEEE Intell. Syst.*, vol. 22, no. 5, Sept.-Oct. 2007, pp. 84-88, doi:10.1109/MIS.2007.4338499.

[18] A. L. N. Escorcio and J. Cardoso, "Editing Tools for Ontology Creation," in *Semantic Web Services: Theory, Tools and Applications*, IGI Global, 2007, ch. 4.

[19] I. H. Witten and E. Frank, "Introduction to Weka" in *Data Mining: Practical Machine Learning Tools and Techniques*, 2nd ed., San Francisco, CA: Morgan Kaufmann Publishers, 2005, ch. 9, pp. 365-368.

[20] I. H. Witten and E. Frank, "Input: Concepts, Instances, and Attributes" in *Data Mining: Practical Machine Learning Tools and Techniques*, 2nd ed., San Francisco, CA: Morgan Kaufmann Publishers, 2005, ch. 2, pp. 52-56.

[21] I. H. Witten and E. Frank, "Credibility: Evaluating What's Been Learned" in *Data Mining: Practical Machine Learning Tools and Techniques*, 2nd ed., San Francisco, CA: Morgan Kaufmann Publishers, 2005, ch. 5, pp. 149-151, 168-173.

[22] A. Lamkanfi, S. Demeyer, Q. D. Soetens, and T. Verdonck, "Comparing Mining Algorithms for Predicting the Severity of a Reported Bug," *Proc. 15th European Conf. on Software Maintenance and Reengineering*, Oldenburg, 2011, pp. 249-258, doi:10.1109/CSMR.2011.31.

[23] I. H. Witten and E. Frank, "The Explorer" in *Data Mining: Practical Machine Learning Tools and Techniques*, 2nd ed., San Francisco, CA: Morgan Kaufmann Publishers, 2005, ch. 10, pp. 403-414.

[24] I. H. Witten and E. Frank, "Algorithms: The Basic Methods" in *Data Mining: Practical Machine Learning Tools and Techniques*, 2nd ed., San Francisco, CA: Morgan Kaufmann Publishers, 2005, ch. 4, pp. 88-92, 121-124.

[25] X. Wu, V. Kumar, J. R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, P. S. Yu, Z.-H. Zhou, M. Steinbach, D. J. Hand, and D. Steinberg, "Top 10 algorithms in data mining," *J. Knowledge and Inform. Syst.*, vol. 14, no. 1, Dec. 2007, pp. 1-37, doi:10.1007/s10115-007-0114-2.

Appendix A

The reason for the construction of sub-rule R3.2 in Section 3.1.5.

In order to reveal the reason for the construction of sub-rule R3.2 we will look into the negation of Rules 1, 2, 4 and 5. Since Rules 1, 4 and 5 are easy to understand and have few sub-rules, it is straightforward to find their negated versions. The challenge is to negate Rule 2. If we denote with A the set of defects that satisfies Rule 2, then the result of negating Rule 2 will yield set B, where B = not A. Then, it is clear that the union of A and B must be equal to 1 or, in other words, A or B = 1.

However, it turns out that, because of the complexity of Rule 2 and the way it is constructed, A or B ≠ 1. This is shown through an example.

Sub-rules R2.2, R2.3 and R2.4 are considered for simplicity (the example will follow the same reasoning if all sub-rules are considered). The available options when considering these three sub-rules are listed below.

- If (hasEffectOnNumber exactly 2) is true (sub-rule R2.2), then:
  - If (isInserted only (InDesign or InRequirements)) is true (sub-rule R2.3), the defect can have one, or two, or all three of the values Data, Interface and Logic of attribute Type (sub-rule R2.4).
  - If ((isInserted only (InCoding or InConfiguration)) is true and (hasType min 2) is true (sub-rule R2.3), the defect can have two or all three of the values Data, Interface and Logic of attribute Type (sub-rule R2.4).
- If (hasEffectOnNumber exactly 3) is true (sub-rule R2.2), then:
  - If ((isInserted only (InDesign or InRequirements)) is true (sub-rule R2.3), the defect can have one, or two, or all three of the values Data, Interface and Logic of attribute Type (sub-rule R2.4).
  - If ((isInserted only (InCoding or InConfiguration)) is true and (hasEffectOnNumber exactly 3) is true while (hasType min 2) is false (sub-rule R2.3), the defect can have exactly one of the values Data or Interface or Logic of attribute Type (sub-rule R2.4).
  - If ((isInserted only (InCoding or InConfiguration)) is true and (hasEffectOnNumber exactly 3) is true while (hasType min 2) is also true (sub-rule R2.3), the defect can have two or all three of the values Data, Interface and Logic of attribute Type (sub-rule R2.4).

After a close observation of these options, it becomes obvious that the following set of defects (denoted with C) is not part of set A: defects that are inserted during the coding phase or the configuration phase and are affecting exactly two values of attribute Effect and exactly one of the values Data or Interface or Logic of attribute Type. This set is not present for a reason. Rule 2 is constructed not to include it. Hence, one would expect that set C will be a subset of set B.

Now, Rule 2 is negated and we take a look particularly at the negated version of sub-rule R2.4. This sub-rule becomes hasType some (not Data and not Interface and not Logic). This means that the available options for the values of attribute Type do not include Data, Interface and Logic.

Therefore, set C, discussed above, will not be part of set B because set B does not permit defects which have any of the values Data, Interface or Logic of attribute Type. This clearly shows that A or B ≠ 1 but that A or B or C = 1. For this reason, the fragment

((isInserted only (InCoding or InConfiguration)) and (hasEffectOnNumber exactly 2) and ((hasType only Data) or (hasType only Interface) or (hasType only Logic))),

which represents the defects from set C, was added to sub-rule R3.2 when it was constructed. The sub-rule is presented below, as given in Section 3.1.5.

(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

Appendix B

Table B.1.L. The information about the selected defects (from the project in CS1) converted into the attributes and their values from the IEEE Standard in [2]

Defect ID in the ontology	*Effect*	*Type*	*Insertion Activity*	*Detection Activity*
101	Functionality; security; performance; serviceability	Data; interface	Design	Supplier testing
102	Usability; performance	Logic	Coding	Supplier testing
103	Functionality; performance	Logic	Design	Supplier testing
104	Usability; performance	Interface	Design	Supplier testing
105	Functionality; performance	Logic	Coding	Supplier testing
106	Usability; performance	Interface	Design	Supplier testing
107	Functionality; performance	Logic	Coding	Supplier testing
108	Functionality; security; serviceability	Data; logic	Coding	Supplier testing
109	Usability; performance	Interface; logic	Coding	Supplier testing
110	Functionality; performance	Data; logic	Configuration	Coding
111	Functionality; serviceability	Data	Requirements	Supplier testing
112	Usability	Interface	Requirements	Supplier testing
113	Usability; performance	Data	Design	Supplier testing
114	Usability; performance	Interface	Design	Supplier testing
115	Functionality; performance	Logic	Coding	Supplier testing
116	Functionality; performance	Data	Requirements	Supplier testing
117	Functionality; serviceability	Data; logic	Design	Supplier testing
118	Functionality; security; performance	Logic	Coding	Supplier testing
119	Usability	Data	Requirements	Supplier testing

Table B.1.R. Severity levels of the selected defects

Taken from the project	*Converted to the IEEE Std*	*Predicted by MAPDESO*
Show-stopper	Blocking	Blocking
Severe	Critical	Major
Severe	Critical	Critical
Severe	Critical	Critical
Severe	Critical	Major
Severe	Critical	Critical
Severe	Critical	Major
Severe	Critical	Critical
Severe	Critical	Critical
Severe	Critical	Critical
Severe	Critical	Critical
Medium	Major	Inconse-quential
Medium	Major	Critical
Medium	Major	Critical
Medium	Major	Major
Medium	Major	Critical
Medium	Major	Critical
Medium	Major	Critical
Medium	Major	Inconse-quential

120	Functionality; performance	Logic	Coding	Supplier testing	Medium	Major	Major
121	Usability; serviceability	Data	Design	Supplier testing	Medium	Major	Critical
122	Functionality; performance	Logic	Coding	Supplier testing	Medium	Major	Major
123	Performance; serviceability	Standards	Requirements	Supplier testing	Medium	Major	Major
124	Functionality; performance	Logic	Coding	Supplier testing	Medium	Major	Major
125	Functionality	Logic	Coding	Coding	Medium	Major	Minor
126	Functionality; performance	Logic	Design	Supplier testing	Medium	Major	Critical
127	Functionality	Data	Coding	Supplier testing	Medium	Major	Minor
128	Functionality; performance	Logic	Coding	Supplier testing	Medium	Major	Major
129	Usability	Standards	Requirements	Supplier testing	Minor	Minor	Inconse-quential
130	Functionality	Data	Requirements	Other	Minor	Minor	Minor
131	Usability	Interface	Design	Supplier testing	Minor	Inconse-quential	Inconse-quential
132	Functionality	Logic	Coding	Coding	Minor	Minor	Minor
133	Usability	Interface	Design	Supplier testing	Minor	Inconse-quential	Inconse-quential

Table B.2.L. The information about the selected defects (from the project in CS2) converted into the attributes and their values from the IEEE Standard in [2]

Defect ID in the ontology	*Effect*	*Type*	*Insertion Activity*	*Detection Activity*
201	Functionality; usability; performance; serviceability	Data; logic	Coding	Customer testing
202	Functionality; usability; performance	Logic	Coding	Coding
203	Functionality; usability; performance	Logic	Coding	Coding
204	Functionality; usability; performance	Logic	Coding	Customer testing
205	Functionality; usability; performance	Logic	Coding	Production
206	Functionality; usability; performance	Data; logic	Coding	Customer testing
207	Functionality; usability; performance	Logic	Coding	Production
208	Functionality; performance	Logic	Coding	Production
209	Functionality; usability; performance	Logic; interface	Coding	Production
210	Functionality; performance	Logic	Coding	Coding
211	Functionality; usability; performance	Logic	Coding	Production
212	Functionality; usability; performance	Data; logic	Coding	Coding
213	Functionality; usability	Logic	Coding	Production
214	Functionality; performance	Interface; logic	Coding	Customer testing
215	Functionality; performance	Logic	Design	Production
216	Functionality; performance	Data	Coding	Production
217	Functionality; performance	Data	Coding	Customer testing
218	Functionality; usability; performance	Interface; logic	Design	Production
219	Functionality; usability	Logic	Coding	Customer testing
220	Functionality; performance	Logic	Coding	Coding
221	Functionality; performance	Logic	Coding	Customer testing
222	Functionality; performance	Data; logic	Design	Customer testing
223	Usability; performance	Logic	Coding	Production
224	Functionality	Logic	Coding	Production
225	Functionality; usability	Logic	Coding	Production

Table B.2.R. The severity levels of the selected defects

Taken from the project	*Converted to the IEEE Std*	*Predicted by MAPDESO*
Block	Blocking	Blocking
Crash	Critical	Critical
Crash	Critical	Critical
Crash	Critical	Critical
Crash	Critical	Critical
Crash	Critical	Critical
Crash	Critical	Critical
Crash	Critical	Major
Crash	Critical	Critical
Crash	Critical	Major
Crash	Critical	Critical
Crash	Critical	Critical
Major	Major	Major
Major	Major	Critical
Major	Major	Critical
Major	Major	Major
Major	Major	Major
Major	Major	Critical
Major	Major	Major
Major	Major	Major
Major	Major	Major
Major	Major	Critical
Minor	Minor	Major
Minor	Minor	Minor
Minor	Minor	Major

226	Functionality	Logic	Coding	Coding	Minor	Minor	Minor
227	Functionality	Logic	Coding	Coding	Minor	Minor	Minor
228	Functionality; usability	Logic	Coding	Production	Minor	Minor	Major
229	Functionality; performance	Logic	Coding	Production	Minor	Minor	Major
230	Usability	Logic	Coding	Production	Minor	Minor	Inconse-quential
231	Performance	Interface	Coding	Production	Minor	Minor	Minor
232	Usability	Logic	Coding	Production	Minor	Minor	Inconse-quential
233	Functionality; usability	Logic	Coding	Production	Minor	Minor	Major
234	Functionality; performance	Logic	Coding	Production	Minor	Minor	Major
235	Functionality; usability	Logic	Coding	Supplier Testing	Minor	Minor	Major
236	Functionality; performance	Data	Coding	Production	Minor	Minor	Major
237	Usability	Data	Coding	Customer testing	Minor	Inconse-quential	Inconse-quential
238	Usability	Interface	Design	Production	Minor	Minor	Inconse-quential
239	Functionality; performance	Data	Coding	Production	Minor	Minor	Major
240	Performance	Interface	Coding	Production	Minor	Minor	Minor
241	Functionality; performance	Logic	Coding	Production	Minor	Minor	Major
242	Performance	Interface	Coding	Coding	Minor	Minor	Minor
243	Performance	Data	Coding	Production	Minor	Minor	Minor
244	Usability	Logic	Coding	Coding	Minor	Minor	Inconse-quential
245	Usability	Interface	Design	Coding	Minor	Minor	Inconse-quential
246	Usability	Interface	Coding	Coding	Minor	Minor	Inconse-quential
247	Usability	Logic	Coding	Coding	Minor	Minor	Inconse-quential

Table B.3.L. The information about the selected defects (from the project used for the validation) converted into the attributes and their values from the IEEE Standard in [2]

Defect ID in the ontology	*Effect*	*Type*	*Insertion Activity*	*Detection Activity*
301	Functionality; performance; serviceability	Logic	Coding	Production
302	Performance; serviceability	Other	Coding	Production
303	Functionality; performance; serviceability	Logic	Coding	Production
304	Functionality; performance; serviceability; usability	Logic	Coding	Production
305	Serviceability	Other	Configuration	Production
306	Functionality; serviceability	Logic; data	Coding	Production
307	Functionality; usability	Other	Coding	Production
308	Functionality; usability	Logic	Coding	Production
309	Functionality	Other	Configuration	Production
310	Functionality; usability; performance	Logic	Coding	Production
311	Performance; usability	Other	Design	Production
312	Performance; functionality; usability	Logic	Coding	Production
313	Functionality; performance; serviceability	Logic	Coding	Production
314	Functionality; serviceability	Other	Configuration	Production
315	Functionality; serviceability	Data; logic	Configuration	Production
316	Functionality	Logic; interface	Configuration	Customer testing
317	Functionality; performance	Interface	Configuration	Production
318	Performance; functionality; usability	Data	Configuration	Production
319	Functionality; performance	Logic	Coding	Production
320	Functionality; serviceability	Interface; logic	Coding	Supplier testing
321	Functionality; performance; usability	Logic	Coding	Production
322	Functionality	Logic	Coding	Production
323	Functionality; usability	Logic	Configuration	Production

Table B.3.R. The severity levels of the selected defects

Taken from the project	*Converted to the IEEE Std*	*Predicted by MAPDESO*
Medium	Major	Critical
Medium	Major	Major
Medium	Major	Critical
Top	Blocking	Blocking
Medium	Major	Minor
High	Critical	Critical
Medium	Major	Major
Low	Minor	Major
Low	Minor	Minor
Low	Minor	Critical
Medium	Major	Major
Medium	Major	Critical
Medium	Major	Critical
Medium	Major	Major
High	Critical	Critical
Medium	Major	Minor
Low	Minor	Major
High	Critical	Critical
Medium	Major	Major
High	Critical	Critical
High	Critical	Critical
Low	Minor	Minor
High	Critical	Major

324	Functionality	Logic; interface	Configuration	Production	Low	Minor	Minor
325	Functionality; usability	Data	Configuration	Production	Medium	Major	Major
326	Functionality; performance	Logic	Coding	Production	Low	Minor	Major
327	Functionality; security; performance; serviceability	Interface; logic	Configuration	Production	Top	Blocking	Blocking
328	Functionality; performance	Logic	Coding	Production	Low	Minor	Major
329	Functionality	Logic	Coding	Production	Low	Minor	Minor
330	Performance; functionality	Data	Configuration	Production	Medium	Major	Major
331	Functionality	Logic	Coding	Production	Low	Minor	Minor
332	Functionality; usability	Logic	Coding	Production	Low	Minor	Major
333	Functionality; serviceability	Logic	Coding	Production	Medium	Major	Major
334	Functionality; serviceability	Logic	Coding	Production	Medium	Major	Major
335	Serviceability	Other	Configuration	Production	Low	Minor	Minor
336	Serviceability; functionality	Logic	Coding	Production	Low	Minor	Major
337	Serviceability	Interface	Configuration	Production	Low	Minor	Minor
338	Serviceability	Interface	Configuration	Production	Low	Minor	Minor
339	Serviceability	Logic	Configuration	Production	Low	Minor	Minor
340	Serviceability	Other	Configuration	Production	Low	Minor	Minor
341	Usability	Logic	Coding	Production	Low	Minor	Inconseq
342	Serviceability	Data	Configuration	Production	Low	Minor	Minor
343	Functionality; usability	Logic	Coding	Production	Low	Minor	Major
344	Performance; functionality; usability	Logic	Coding	Production	High	Critical	Critical
345	Functionality; serviceability	Logic	Coding	Production	Low	Minor	Major
346	Functionality; performance	Interface	Design	Production	Medium	Major	Critical
347	Serviceability	Data	Coding	Production	Low	Minor	Minor
348	Functionality; performance	Interface; data	Configuration	Production	High	Critical	Critical
349	Functionality	Logic	Coding	Production	Low	Minor	Minor
350	Functionality; performance; usability	Interface	Configuration	Production	High	Critical	Critical