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

Karasneh, B.H.A.

Chapter 3

A Method for Automated Prediction of Defect Severity Using Ontologies

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

This chapter is based on the following publication:

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

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

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

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

3.1 Approach

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

3.1.1 Developing the Ontology

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

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

3.1.1.1 Meta-meta Level

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

3.1.1.2 Meta Level

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

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

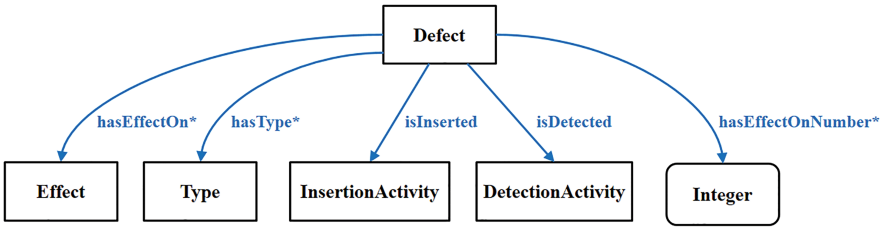

Figure 3.1: *The created classes and properties for the ontology*

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

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

3.1.1.3 Class Level

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

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

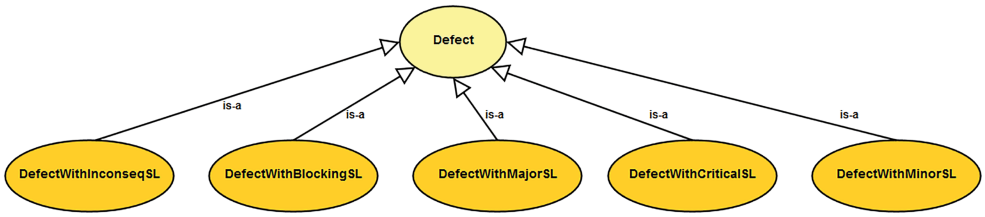

Figure 3.2: *Class Defect, its subclasses*

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

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

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

3.1.1.4 Instance Level

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

3.1.1.5 Classification Rules

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

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

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

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

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

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

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

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

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

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

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

3.1.2 The Method Flow

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

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

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

3.2 Case Studies

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

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

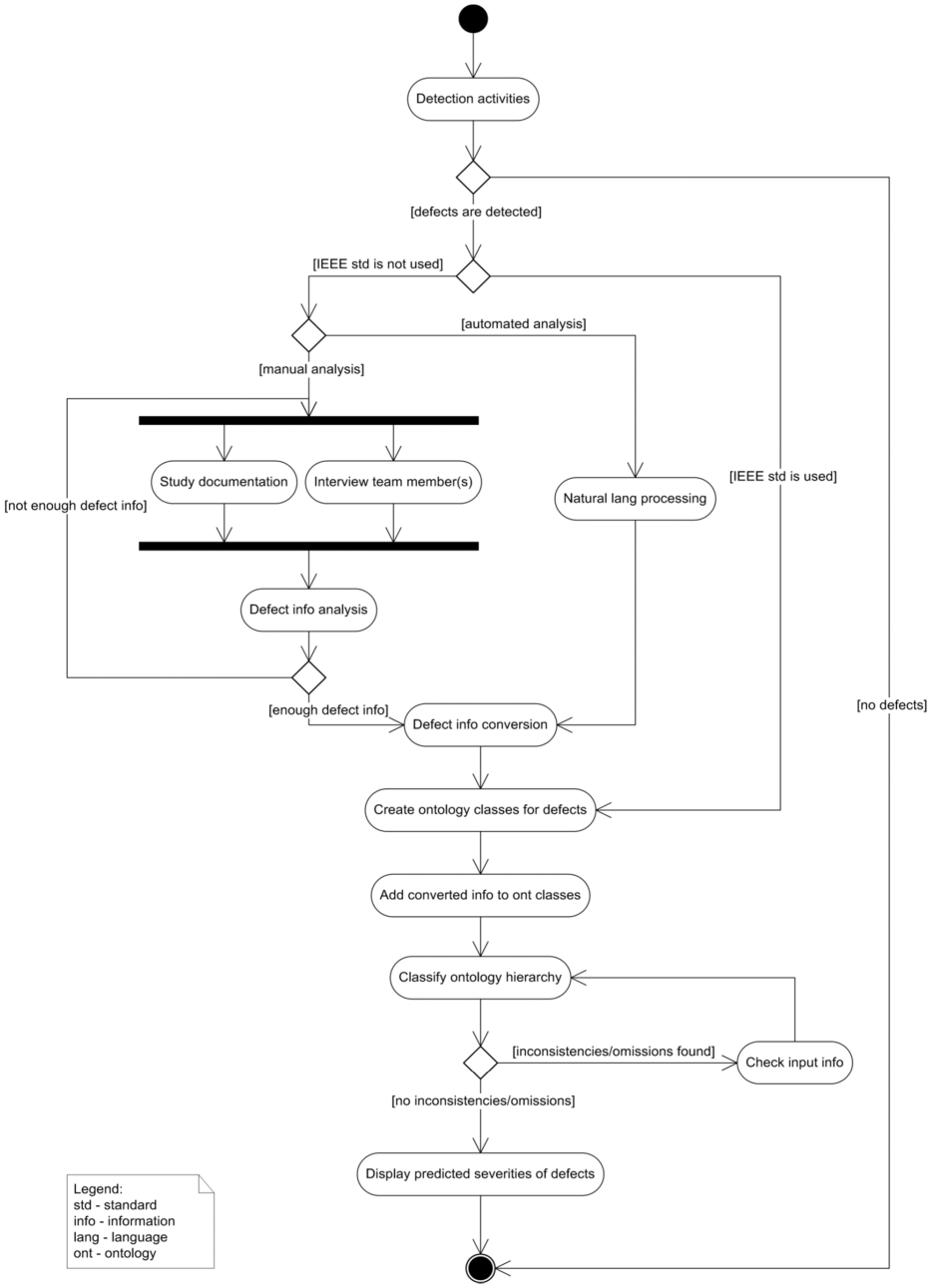

Figure 3.3: *Activity diagram for the prediction of defects' severity levels*

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

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

3.2.1 Data Collection

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

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

3.2.2 Data Analysis and Conversion

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

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

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

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

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

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

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

3.2.3 Data Classification

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

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

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

3.2.4 Results and Comparison

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

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

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

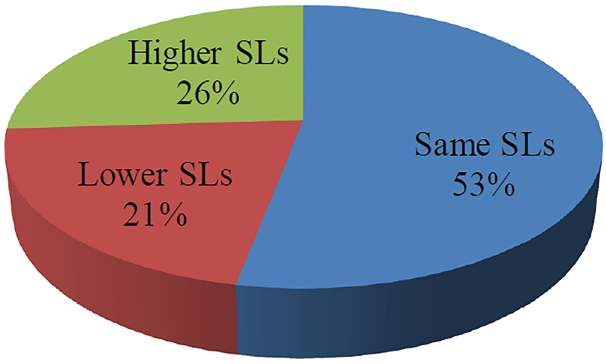

Figure 3.4: *Percentages of the 80 defects classified into the same severity levels (SLs), lower SLs and higher SLs by the ontology compared with the original classifications from CS1 and CS2*

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

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

3.3 Validation

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

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

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

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

3.3.1 Approach - VCS

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

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

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

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

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

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

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

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

3.3.2 Results and Comparison

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

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

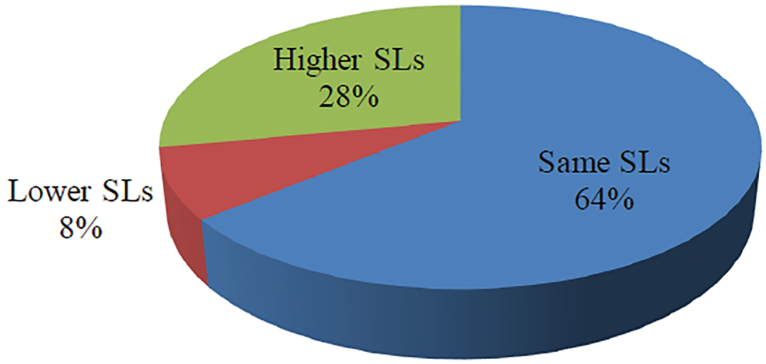

Figure 3.5: *Percentages of the 50 defects classified into the same severity levels (SLs), lower SLs and higher SLs by the ontology compared with the original classification from VCS*

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

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

3.3.3 Validation of the Results

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

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

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

3.3.4 Comparison

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

3.3.4.1 Predicting severities of defects using Weka classifiers

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

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

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

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

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

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

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

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

3.3.4.2 Comparison of the performances

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

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

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

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

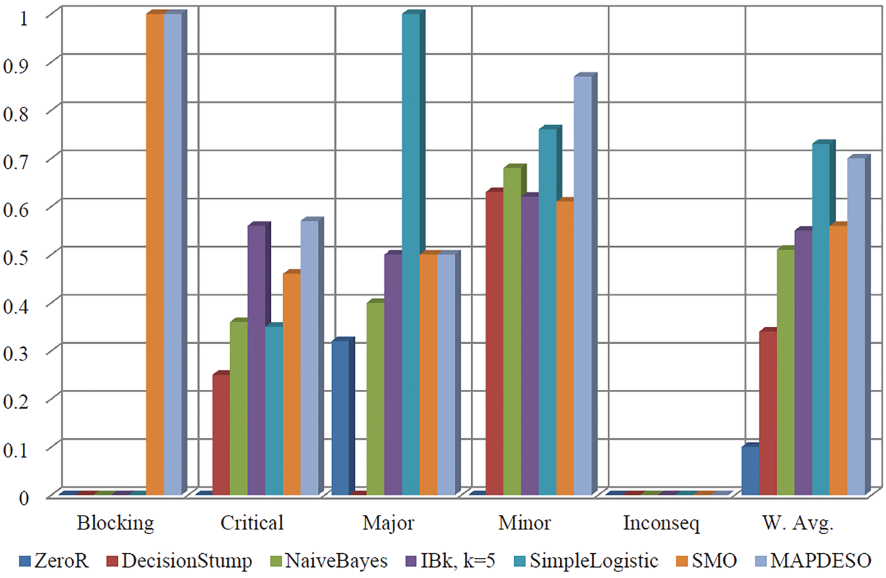

Figure 3.6: *The results for precision per severity level for the six classifiers and for MAPDESO*

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

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

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

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

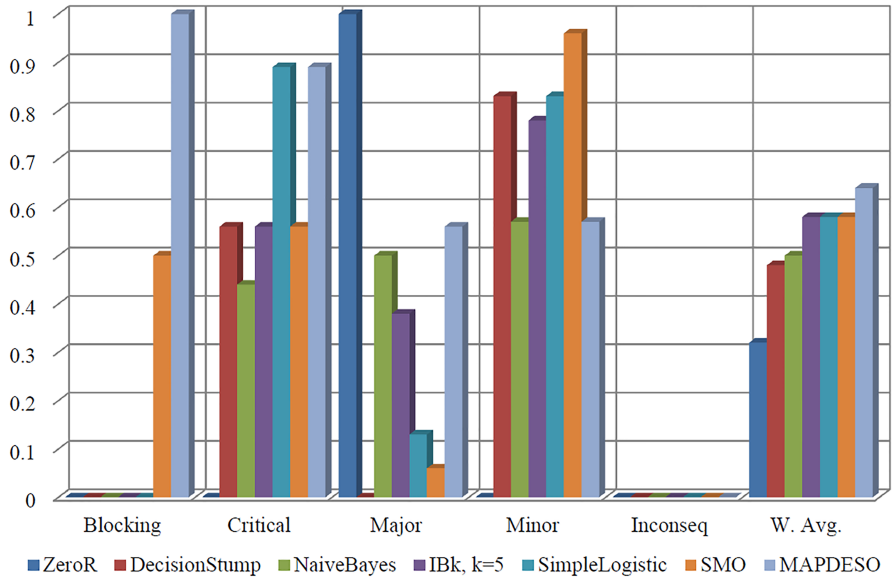

Figure 3.7: *The results for recall per severity level for the six classifiers and for MAPDESO*

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

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

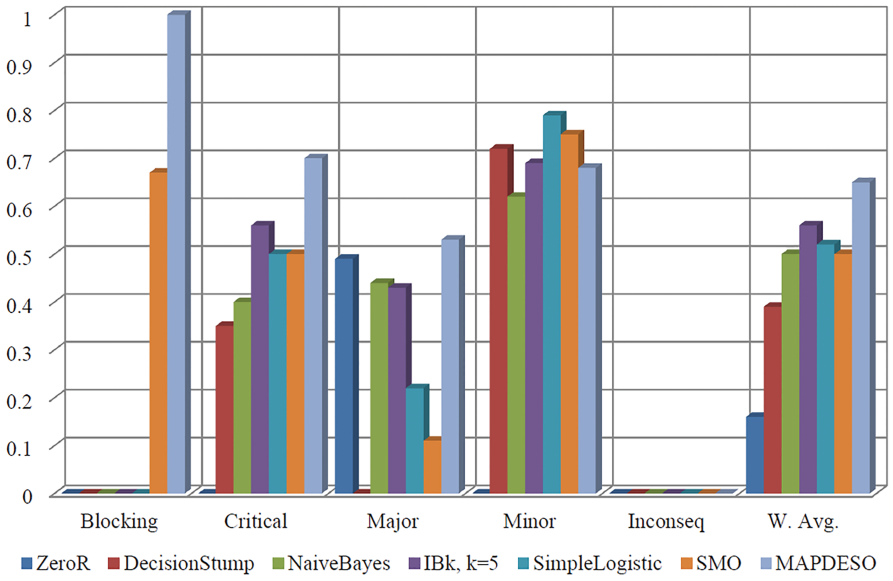

Figure 3.8: *The results for F-measure per severity level for the six classifiers and for MAPDESO*

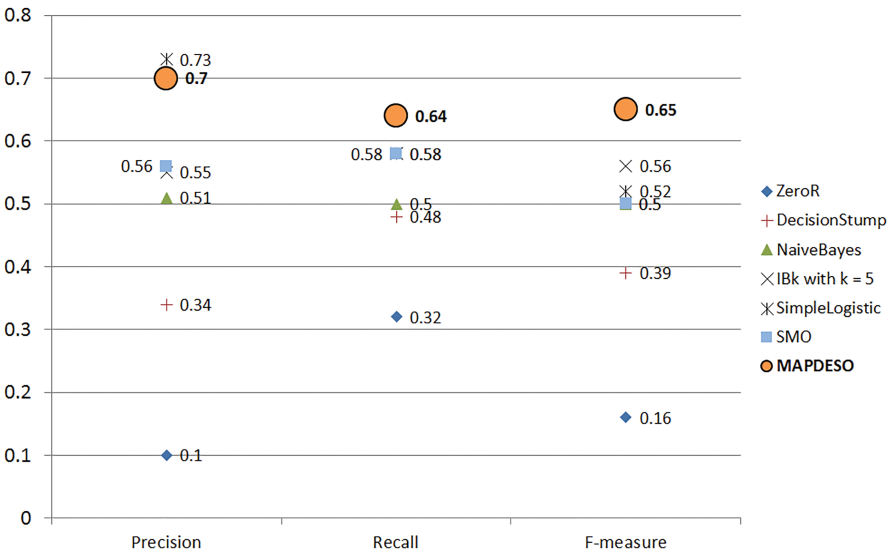

Figure 3.9: *Summary of the comparison between the six classifiers and MAPDESO*

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

3.4 Related Work

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

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

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

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

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

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

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

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

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

3.5 Threats to Validity

In this section we discuss threats to validity.

3.5.1 Conclusion Validity

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

3.5.2 Internal Validity

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

3.5.3 Construct Validity

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

3.6 Conclusions

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

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

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

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

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

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

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

3.7 Future Work

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

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