Interactive scalable condensation of reverse engineered UML class diagrams for software comprehension

Osman, M.H.B.

Chapter 8

Condensing Reverse Engineered Class Diagrams through Class Name Based Abstraction

In this chapter, we report on a machine learning approach to condensing class diagrams. This research focuses on building a classifier that is based on the names of classes in addition to design metrics, and we compare to earlier work that is based on design metrics only. We assess our condensation method by comparing our condensed class diagrams to class diagrams that were made during the original forward design. Our results show that combining text metrics with design metrics leads to modest improvements over using design metrics only. On average, the improvement reaches 5.3%. 7 out of 10 evaluated case studies show improvement ranges from 1% to 22%.

8.1 Introduction

In our previous work (Chapter 7), we proposed an approach to simplify RE-CDs based on static analysis. We used forward designs as 'ground truth' for what classes are most important, and then used machine learning techniques to learn the relationship between class characteristics measured in object-oriented design metrics, and the importance of the class (in forward design or not). The classifier model delivers a score

This chapter is adapted from a publication entitled **"Condensing Reverse Engineered Class Diagrams through Class Name Based Abstraction"**, 2014 World Congress on Information and Communication Technologies (WICT)

for each class that indicates the importance (likelihood to be in the forward design), so that developers can explore the class diagrams at different levels of abstraction by varying the threshold on the score.

In this chapter, we extend the previous work by introducing text based metrics derived from class names. From our previous survey (Chapter 5), we found that software engineers consider both design metrics (e.g. Coupling, Number of methods) and class element names (i.e. Related to domain) to be key features to decide on the importance of a particular class. The previous study (Chapter 7) is used as a baseline and we also follow the experimental setup for this research. We derive several text metrics from the set of class names. We then apply a set of classification algorithms to the text metrics and compare the result with the design metrics and a combination of both design and text metrics, using the Area Under the Curve (AUC) evaluation measure.

The contributions of this study are the following:

- Formulation of text metrics based on class names for classification of key classes.
- Combining text metrics and design metrics for classification of key classes.
- Evaluation of the approach shows that the combination of text and design metrics has improved the prediction performance.

This chapter is structured as follows: Section 8.2 discusses the related work. Section 8.3 describes the research questions. Section 8.4 explains the approach while Section 8.5 explains the experiment description. We present the analysis of results in Section 8.6 and discuss our findings in Section 8.7. This followed by conclusions and future work in Section 8.8.

8.2 Related Work

In this section, we discuss related work. We consider studies on the usage of text in software documentation and we found the research in the areas of code summarization and analysis of execution traces are related to our topic.

8.2.1 Code Summarization

Code summarization is an approach to summarize source code to help program comprehension. Haiduc et al. [74] introduced automated source code summarization based on the text retrieval approach. Their research used the natural language summarization technique to summarize the source code using lexical (i.e. identifiers and comments) and structural information (i.e. class, method, function). They generated a list of important keywords of methods. The generated important keywords were validated by six software developers. This preliminary study found that the method's names

influence the identification of important keywords in a method. Their study showed that 98.7% of identified keywords are derived from methods' name.

Haiduc et al. [75] extended the research by evaluating four text summarization techniques for the purpose of code summarization. For this evaluation, they generated various formations of text (e.g. by using weight – *tf-idf, binary-entropy, log*), as well as different types of the text summarization techniques. The generated keywords were validated by four software developers. They discovered that the combination of text summarization technique (i.e. Lead+Vector Space Model) performed the best result in capturing the meaning of methods and classes in an object-oriented source code. It is interesting to see that their research also indicates that class names are essential information in summarizing source code. This research has been replicated and extended by Eddy et al. [50]. Eddy et al. extended the work by evaluating a text retrieval technique (Hierarchical Pachinko Allocation Model). The validation was performed by 14 software developers, and their results confirmed Haiduc et al.'s findings.

Moreno et al. [114] proposed an automatic structured natural language summaries generator specifically for Java classes. Differently from the work by Haiduc et al., they use detailed structural class information (i.e. class stereotype) in addition to the class identifier to generate a summary of classes. Their validation illustrates that 90% of the generated class summaries were concise, readable and understandable. 69% of the summaries did not miss the important information.

Our study shares with text summarization that we look for cues of importance in the (key)words used in classes and method. Rather than aiming for a textual summarization, we target graphical summaries. A difference between our work and the text-based summarization approaches is that text-based summarization does not take information about class relations into account. From that perspective, our approach is more aimed at summarizing a design, while the text-based summarization is more focussed on summarization of individual classes (which may be used to summarize a system when seen as a set of classes). However, our work could benefit from mechanisms for selecting important information of classes for inclusion in class diagrams. We keep this as a proposal for future work.

8.2.2 Analysis of Execution Trace

Pirzadeh et al. [138] proposed a technique to simplify the analysis of execution traces for understanding a system's behavior. Their technique analyses execution traces by analyzing the most relevant information about the execution traces according to the execution phases.

Medini et al. [111] presented the Segment Concept AssigNer (SCAN) to assign labels to sequences of methods and to discover relations between segments by using execution traces as input. They utilized method-names, parameters and code-bodies. They applied text processing to filter the collected text.

Lin et al. [105] proposed Unsupervised Induction of Concepts (UNICON) to cluster large numbers of elements of semantically related words using an unsupervised technique. The inputs of their study are: (1) a collocation database and, (2) a similarity matrix of words. They used a clustering algorithm to break up a large set of data into small subsets. Then, they apply UNICON to suggest a set of concepts of the word.

The related works discussed above use text derived from the execution trace to understand the system. In contrast, we use the information that can be extracted from the source code and design documentation (static analysis). The volume of text derived from execution traces is so huge that the research in that line may only group the execution in phases. However, we acknowledge the use of information derived from execution traces as an interesting complement to our approach.

8.3 Research Questions

This section describes the main research question and sub-questions. The main question of this research is the following:

MQ: *Do text metrics derived from class names add value over design metrics to identify key classes to be included in a class diagram?*

In order to answer this question, the answers to the following questions need to be explored:

- **RQ1:** *What is the performance of text predictors?*
- **RQ2:**. *What are the most influential text predictor(s)?*
- **RQ3:**. *What is the performance of the classification algorithms in using text metrics as predictors?*
- **RQ4:**. *Which set of predictors produces the best result compared to design metrics?*

The research questions are answered in Section 8.6 and the main question is answered in Section 8.8.

8.4 Approach

In this section, we describe our approach in conducting this experiment. The overall framework of this experiment is shown in Figure 8.1. The input for this study are the system documents (*Step 1*); which consist of the RE-CD in XML Metadata Interchange (XMI) format and the Forward Design (FD). Section 8.4.1 describes the inputs in detail. Then, we normalize the text (class names) by removing the non-informational characters and words (*Step 2*). This includes converting class names into streams of words. In total, there are four subprocess involved, which are Lexical Analysis,

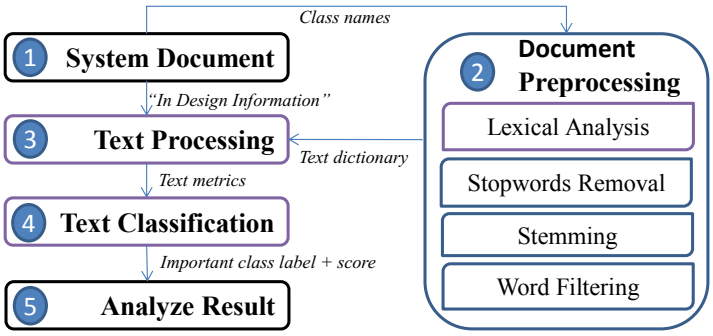

Figure 8.1: *Overall Framework*

Stopwords Removal, Stemming and Single Words Filtering. This process produces a text dictionary (Section 8.4.2 describes this process in detail). We then define the text metrics based on the text dictionary (*Step 3*). As a result, nine text metrics are invented. The detailed information about the text metrics is described in Section 8.4.3. For text classification (*Step 4*), we evaluate nine classification algorithms. Detailed information about this is described in Section 8.4.4. Finally, we analyse the important classes and classification score (*Step 5*).

8.4.1 System Document

A "raw" RE-CD is generated based on source code by using MagicDraw [9] (version 17.0). The class names in RE-CD will be used to form the predictors. Forward designs (FD) are the designs that were created during the development of the system. We assume all designs in the project's documentation are forward design. The machine learning algorithms use the FDs to learn the positive instances. The list of datasets (OSSD projects) is shown in Table 8.1. The datasets were collected from different types of domain. The detailed information about these projects can be found at Chapter 3.

8.4.2 Document Preprocessing

For data preparation, we use an automatic indexing procedure for English as mentioned in [175] and also the document preprocessing procedure suggested by [18]. This procedure includes lexical analysis, stop words removal, words filtering and stemming. The processes of this procedure are the following.

Lexical Analysis of the Text

Lexical analysis refers to a technique of converting a stream of characters into a stream of words [18]. One of the objectives of this technique is to identify the word from the class names collection. From our observation, most of the class names consisted of one

Table 8.1: *List of Case Study*

No.	Project	Total Classes in Re-CD **(S)**	Total Classes in FD **(D)**	D:S **ratio (%)**
1.	ArgoUML	903	44	4.87
2.	Mars	840	29	3.45
3.	JavaClient	214	57	26.64
4.	gwt-Portlets	178	20	11.24
5.	JGAP	171	18	10.52
6.	Neuroph	161	24	14.90
7.	JPMC	121	24	19.83
8.	Wro4J	87	11	12.64
9.	xUML	84	37	44.05
10.	Maze	59	28	47.45

or more words. Normally, capital letters are used to separate these words. Therefore, we separated the word(s) in class names from each document based on capital letters. The algorithm for class names separation is presented in Algorithm 1. For instance, the class name "PlayerSimulationData" will be separated into 3 different words: "Player", "Simulation" and "Data". There are also several exceptional cases where class names use multiple capital letters (such as "MMClassKeyword", "GAAlgorithm", "LMS" and "IACNeuron"). This issue is addressed by lines 16-17 in Algorithm 1. Then, the text documents generated from the lexical analysis process were loaded into a text processing tool. RapidMiner version 5.3 [11] was used as the text processing tool.

Stop Words Removal

This study requires only significant (meaningful) keywords. Stop words such as "the" or "and" help to build ideas, but do not carry any significance themselves [143]. Hence, these words are removed from the keywords' list. For example, the class name "PlantsAndAnimal" consists of the following words: "Plants", "And", and "Animal". The keywords "Plants" and "Animal" are meaningful keywords while the word "And" is only used to connecting those keywords. This process was done automatically by using the text processing tool (RapidMiner).

Stemming

A lot of words have the same root meaning but appear as different variants. A word for instance "maintenance" and "maintain" have the same word root. Hence, the stemming algorithm is used to resolve this issue. Stemming refers to a computational procedure that reduces all words with the same root (or, if prefixes are left untouched, the same stem) to a common form, usually by stripping each word of its derivational

Algorithm 1 Class Name Separation Algorithm

```
1:  Input :
2:  Document (D) = list of class names
3:  Output :
4:  Names = list of words
5:  Method :
6:  ClsNameChar = array of characters
7:  while D not End of List do
8:     Convert class names to array of characters (AC)
9:     for count = 0 to AC.size do
10:       if count = 0 and AC[count] is Uppercase then
11:          Add AC[count] to ClsNameChar {#Begin of a word}
12:       else if AC[count] is Uppercase and AC[count -1] is Lowercase then
13:          Add ClsNameChar to Names {#A word is completed}
14:          Empty ClsNameChar
15:          Add AC[count] to ClsNameChar {#Begin of new word}
16:       else if AC[count] is Uppercase and AC[count + 1] is UpperCase then
17:          Add AC[count] to ClsNameChar {#For words that use only capital letters}
18:       else
19:          Add AC[count] to ClsNameChar
20:       end if
21:    end for
22: end while
23: return Names
```

and inflectional suffixes [106]. For this, we used the Porter stemming algorithm [140] provided by the text processing tool (RapidMiner).

Word Filtering

Normally, a word consists of more than two characters. Hence, we remove all the single character words.

8.4.3 Text Processing

The results of the previous process allow us to create a text dictionary [122]). The text dictionary is formulated based on the following information:

- *NumKeyword*: Number of keyword(s) in a class name.
- *InDoc*: Number of documents (projects) where keywords occur.
- *TotalOcc*: The number of occurrences of words in all documents.
- *SpecOcc*: The number of occurrences of words in a specific document.

We formulated the text metrics based on the assumption that classes that have a lot of frequently used keywords most likely are candidate for classes that should be included (key classes) in the system. We make this assumption as we observe that many important classes indicated in the case studies documents consist of popular (or common) keywords. The list of common and uncommon words are illustrated in Table 8.2 and Table 8.3 respectively. The complete tables of common and uncommon words can be found at [123]. The formulation of the text metrics is the following:

$$K = \{k_1..k_n\} \text{ is a set of } n \text{ keywords} \tag{8.1}$$

$$F = \{f_1..f_m\} \text{ is a set of } m \text{ documents} \tag{8.2}$$

$$TotalOcc(k, f) = \text{ Number of Occurance of } k \text{ in } f \tag{8.3}$$

1. **NumKeyword** is the number of words in a class name. This metrics is gathered by simply calculating the number of word(s) in a class name after the lexical analysis process. The *NumKeyword* is represented as *K* in equation 8.1.

2. **ExiInDoc** counts the presence of a word in all documents. Therefore, *ExiInDoc* is defined as follows:

$$ExiInDoc(k, f) = \begin{cases} 1 & TotalOcc(k, f) > 0 \\ 0 & \text{Otherwise} \end{cases} \tag{8.4}$$

   Hence, we define presence function for a set of documents as:

$$ExiInDoc(k, F) = \sum_{f \in F} ExiInDoc(k, f) \tag{8.5}$$

   And therefore, we have:

$$ExiInDoc(K, F) = \sum_{k \in K} ExiInDoc(k, F) \tag{8.6}$$

   For instance, the class name "AdminConsoleDisplay" consists of: "Admin", "Console", and "Display". "Admin" appears in 3 documents, "Console" appears in 4 documents and "Display" appears in 6 documents. Hence, *ExiInDoc* for "AdminConsoleDisplay" is 13.

3. **MaxInDoc** takes the counts on the presence of a word; which is the highest count of all words in a class name. It is different from *ExiInDoc* where *MaxInDoc* compares the value of every keywords in a class name and takes the highest value. Hence, we define *MaxInDoc* as follows:

$$MaxInDoc(K, F) = \max_{k \in K} ExiInDoc(k, F) \tag{8.7}$$

   By using the example in *ExiInDoc*, the *MaxInDoc* value for "AdminConsoleDisplay" is 6.

4. **TotalOccAll** counts the total number of occurrences of each word in a class name from all documents. Therefore, the *TotalOccAll* for a class name is the following:

$$\mathrm{TotalOccAll}(K, F) = \sum_{k \in K} \mathrm{TotalOcc}(k, F) \tag{8.8}$$

   For example, the number of occurrences in all documents for "AdminConsoleDisplay" is 20 for "Admin", 12 for "Console" and 7 for "Display". Hence, the *TotalOccAll* for "AdminConsoleDisplay" is 39.

5. **MaxOccAll** takes the highest number of occurrences of a word (in all documents) in a class name.

$$\mathrm{MaxOccAll}(K, F) = \max_{k \in K} \mathrm{TotalOcc}(k, F) \tag{8.9}$$

   By using the example in *TotalOccAll* metric, the value of *MaxOccAll* for "AdminConsoleDisplay" is 20.

6. **TotalOccSpec** counts the total number of occurrences of each word in a class name from a specific document.

$$\mathrm{TotalOccSpec}(K, f) = \sum_{k \in K} \mathrm{TotalOcc}(k, f) \tag{8.10}$$

   Let say, the occurrences in a document for "AdminConsoleDisplay" is 10 for "Admin", 6 for "Console" and 2 for "Display". Hence, *TotalOccSpec* is 18.

7. **MaxOccSpec** takes the highest number of occurrences of a word in a class name from a specific document.

$$\mathrm{MaxOccSpec}(K, f) = \max_{k \in K} \mathrm{TotalOcc}(k, f) \tag{8.11}$$

   By using the example in *TotalOccSpec*, the *MaxOccSpec* is 10.

8. **WeightOccAll** is the average value of all word occurrences of a class name in all documents. It took the *TotalAccAll* and divided by the number of *NumKeyword* of a class name. Hence, the calculation *WeightOccAll* is the following:

$$\mathrm{WeightOccAll}(K, F) = \frac{\mathrm{TotalOcc}(K, F)}{\mathrm{NumKeyword}(K, F)} \tag{8.12}$$

9. **WeightOccSpec** is the average value of all word occurrences of a class name in a document. It took the *TotalOccSpec* and divided by the number of *NumKeyword*.

$$\mathrm{WeightOccSpec}(K, f) = \frac{\mathrm{TotalOcc}(K, f)}{\mathrm{NumKeyword}(K, f)} \tag{8.13}$$

Table 8.2: *The Top List of Common Words in Class Diagrams*

No.	word	inDoc	Total Occ	gwt-portlet	Argo UML	Java Client	JGAP	Mars	Neuroph	JPMC	xUML	Maze	Wro4J
1	panel	6	188	12	70	0	0	90	3	12	0	1	0
2	descriptor	2	171	0	1	0	0	170	0	0	0	0	0
3	player	2	153	0	0	152	0	1	0	0	0	0	0
4	model	7	139	0	97	0	0	20	1	6	4	7	4
5	uml	1	105	0	105	0	0	0	0	0	0	0	0
6	list	5	100	2	37	0	1	59	0	0	0	1	0
7	action	3	96	0	94	0	0	0	0	0	1	1	0
8	impl	4	84	7	5	0	0	1	0	0	71	0	0
9	resourc	4	75	0	33	0	0	37	0	1	0	0	4
10	prop	1	75	0	75	0	0	0	0	0	0	0	0
11	data	9	70	11	3	32	5	1	2	11	3	0	2
12	tabl	5	67	0	37	0	1	23	1	5	0	0	0
13	cr	1	67	0	67	0	0	0	0	0	0	0	0
14	tab	2	63	0	24	0	0	39	0	0	0	0	0
15	diagram	2	62	0	53	0	0	0	0	0	9	0	0
16	fig	1	61	0	61	0	0	0	0	0	0	0	0
17	state	4	59	0	50	3	0	0	0	0	5	1	0
18	type	6	55	0	15	0	1	31	4	0	3	0	1
19	config	5	54	1	3	21	8	21	0	0	0	0	0
20	posit	3	52	1	1	50	0	0	0	0	0	0	0
21	event	6	50	6	16	0	2	15	0	6	5	0	0
22	column	2	47	0	46	0	0	1	0	0	0	0	0
23	interfac	2	44	0	10	34	0	0	0	0	0	0	0
24	vehicl	1	43	0	0	0	0	43	0	0	0	0	0

Table 8.3: *The Top List of UnCommon Words in Class Diagrams*

No.	word	inDoc	Total Occ	gwt-portlet	Argo UML	Java Client	JGAP	Mars	Neuroph	JPMC	xUML	Maze	Wro4J
1	zoom	1	1	0	1	0	0	0	0	0	0	0	0
2	zig	1	1	0	0	0	0	0	0	0	0	1	0
3	zero	1	1	0	1	0	0	0	0	0	0	0	0
4	zag	1	1	0	0	0	0	0	0	0	0	1	0
5	yoga	1	1	0	0	0	0	1	0	0	0	0	0
6	xmi	1	1	0	1	0	0	0	0	0	0	0	0
7	write	1	1	0	0	1	0	0	0	0	0	0	0
8	wrapper	1	1	0	0	0	0	0	0	0	0	0	1
9	workout	1	1	0	0	0	0	1	0	0	0	0	0
10	wind	1	1	0	0	0	0	1	0	0	0	0	0
11	win	1	1	0	1	0	0	0	0	0	0	0	0
12	wi	1	1	0	0	1	0	0	0	0	0	0	0
13	weather	1	1	0	0	0	0	1	0	0	0	0	0
14	waypoint	1	1	0	0	1	0	0	0	0	0	0	0
15	wave	1	1	0	0	0	0	0	0	0	0	1	0
16	watch	1	1	0	0	0	0	0	0	0	0	0	1
17	wai	1	1	0	0	0	1	0	0	0	0	0	0
18	void	1	1	0	0	1	0	0	0	0	0	0	0
19	visual	1	1	0	0	0	1	0	0	0	0	0	0
20	violat	1	1	0	1	0	0	0	0	0	0	0	0
21	viewer	1	1	0	0	0	0	0	0	0	0	1	0
22	vi	1	1	0	1	0	0	0	0	0	0	0	0
23	vecmov	1	1	0	0	1	0	0	0	0	0	0	0
24	valid	1	1	0	0	0	1	0	0	0	0	0	0

8.4.4 Text Classification

The text classification process presents the classification activity to find the result of the class inclusion or exclusion in a class diagram. We use the Information Gain (InfoGain) Attribute Evaluator in WEKA [76] to estimate the predictive power of the text predictor. We prepared several sets of predictors. Then, we apply those sets of predictors to all selected classification algorithms to get the AUC score.

8.4.5 Analyze Result

The InfoGain measures and the AUC scores from the text classification process are analysed. We compare the performance of all evaluated classification algorithms across all datasets.

8.5 Experiment Description

This section explains the dataset that we used in this study and the evaluation measure for analyzing the results.

8.5.1 Dataset

All datasets in Chapter 7 were also used for this study. However, we added a project (gwt-Portlets) to the datasets that we found suitable for this study. The list of datasets is shown in Table 8.1. The datasets come from different types of domain. For instance, ArgoUML and xUML come from the UML tool domain, Neuroph comes from the neural network domain and JavaClient comes from the Application Programming Interface (API) domain. The detailed information about these projects can be found at Chapter 3.

The total number of classes in RE-CDs is between 59 and 903. The total number of classes in FDs is between 11 and 57. The ratio between FDs and RE-CDs (D:S ratio) is between 3% to 47%. For instance, let say if the D:S ratio of the JavaClient project is 26.6%, then only 26.6% of the classes that exist in the implementation (and hence in the RE-CD) are also in the FD. Meanwhile, another 73.4% of the classes that exist in the implementation does not appear in the FD. For the purpose of machine learning, only 26.6% of these classes are positive instances and the other 73.4% are negative instances.

With this information (Table 8.1), we see that most of these datasets are imbalanced in positive and negative instances.

8.5.2 Evaluation Measures

This subsection describes the evaluation methods that were used in this study. This evaluation is supported by the Waikato Environment Knowledge Analysis (WEKA)

[76]. The Information Gain Attribute Evaluator analysis [76] is used to evaluate the influence of each individual text metrics in predicting class inclusion/exclusion. This evaluation produces a score from 0 to 1 for every predictor. A value closer to 1 means strong influence.

We used the Area Under the Curve (AUC) [79] for analyzing the performance of the classification algorithm. WEKA provides the AUC calculation that produces a value ranging from 0 to 1. An AUC score approaching 1 means a better classification while a value close to 0.50 means the classifier performs almost randomly. The AUC score is used for evaluation because it may avoid the issue of favouring models that evaluation method just predict the majority outcome class. Because our data is typically imbalanced, such a majority-based score is suitable.

8.5.3 Experiment

This subsection describes the experiment set up in this study. We randomly split 50% the data in a 50% train and 50% test set. As mentioned before, the datasets are typically imbalanced. In the sense that, the classes that are in the RE-CD but not in the FD (the 'negatives') are highly dominated the entire dataset. If we use the 10-fold cross validation, the chance that many positives are included in the test data is very low because it uses 10% of the data for testing and 90% for training. Thus, the test set error measurements would not have been reliable. We ran each of the experiment 10 times using different randomizations to improve the reliability.

8.6 Analysis of Results

This section describes the analysis of results. Every subsection is presented to answer the questions specified in Section 8.3.

8.6.1 RQ1 : Influence of Predictors

We applied the Information Gain Attribute Evaluator analysis to evaluate the influences of individual predictors. The overall results of this evaluation are illustrated in Table 8.4. We consider a predictor as influential when the Information Gain value > 0. The column *Count* counts the number of times that a predictor produces InfoGain values > 0 (means influential) across all ten case studies. Overall, out of ten evaluated case studies, all individual text metrics are influential predictors for at least four case studies. This means all text metrics are capable of influencing the prediction of the important classes in a class diagram.

In terms of the predictive performance across case studies, the gwt-portlets, JPMC, Maze, Neuroph and Wro4j projects recorded poor performance (see Table 8.4). From our observation, the projects produce poor performance due to the fact that there is no

pattern in the class names in the positive instances group for most of the text metrics. Thus, the text metrics could not influence the prediction for these projects.

The predictive performance increases when a case study has several keywords (or *'champion words'*) that frequently appears in classes that grouped in the positive instances. For these classes, the text metrics produced relatively similar values that lead to homogeneity (or a uniform pattern) of data.

8.6.2 RQ2 : Most Influential Predictors

The *TotalOccSpec* and *MaxOccSpec* are the most influential predictors across all case studies. These predictors are influential in six case studies while the *WeightOccSpec* is influential in five case studies (Table 8.4). In addition, Table 8.4 also shows that the average InfoGain values of these three predictors are recorded as among the highest compared to other predictors. The top three most influential predictors come from the text derived from individual case studies. These top three predictors are highly correlated because these are derived from the same base information (*SpecOcc*). These predictors (related to text from a specific project) have higher predictive influence than metrics derived from all together.

8.6.3 RQ3 : Classification Algorithms Performance

In this study, we consider a classification algorithm to be suitable for prediction if its AUC score is above 0.60. Prior study (Chapter 7) shows that the suitable classification algorithm for class inclusion/exclusion based on design metrics are Random Forests and Nearest Neighbor. However, when using just text metrics, Logistic Regressions also provides suitable results. In addition, when design and text metrics are combined, Decision Stumps also perform surprisingly well (Table 8.5). For the xUML and JavaClient projects, AUC scores are higher than 0.60 for all evaluated classification algorithms.

8.6.4 RQ4 : Set of Predictors Performance

In this subsection, we present a comparison between the combination of text and design metrics with the text metrics and the design metrics separately. We also extend the analysis by comparing the sets of results by using the Random Forests classification algorithm that we found performed the best prediction.

Table 8.4: *Information Gain Results for Text Predictors*

No.	Text Metrics	Argo UML	gwt-portlet	Java Client	JGAP	JPMC	Mars	Maze	Neuroph	Wro4J	xUML	Count	Average
1.	TotalOccSpec	0.051	0.000	0.407	0.051	0.000	0.030	0.125	0.000	0.000	0.257	6	0.086
2.	MaxOccSpec	0.048	0.000	0.407	0.085	0.149	0.022	0.000	0.000	0.000	0.146	6	0.092
3.	WeightOccSpec	0.045	0.000	0.341	0.098	0.000	0.018	0.000	0.000	0.000	0.146	5	0.065
4.	TotalOccAll	0.038	0.000	0.512	0.000	0.000	0.030	0.000	0.000	0.000	0.109	4	0.032
5.	MaxOccAll	0.040	0.000	0.407	0.000	0.000	0.027	0.000	0.000	0.000	0.171	4	0.038
6.	WeightOccAll	0.033	0.000	0.259	0.000	0.000	0.025	0.000	0.000	0.110	0.000	4	0.025
7.	MaxInDoc	0.013	0.000	0.076	0.000	0.000	0.019	0.000	0.000	0.000	0.272	4	0.065
8.	NumKeyword	0.016	0.000	0.178	0.000	0.000	0.025	0.102	0.000	0.000	0.000	4	0.069
9.	ExiInDoc	0.013	0.000	0.152	0.000	0.000	0.021	0.000	0.064	0.000	0.000	4	0.043
	No. of InfoGain > 0	**9**	**0**	**9**	**3**	**1**	**9**	**2**	**1**	**1**	**6**		

Table 8.5: *Classification Algorithms Performance on Predictors Sets (AUC Score ≥ 0.60)*

Predictors sets	J48	k-NN (1)	k-NN (5)	Logi. Regr	Naive Bayes	Dec. table	Dec. Stump	RBF	Rand. Forest	Rand. Tree
Text (T)	4	5	9	9	8	2	6	7	9	5
Design (D)	5	8	9	7	8	3	7	7	9	6
Design_Text (DT)	7	6	9	8	9	4	9	9	9	6
Δ(T-D)*	-1	-3	0	2	0	-1	-1	0	0	-1
Δ(DT - T)*	3	1	0	-1	1	2	3	2	0	1
Δ(DT - D)*	2	-2	0	1	1	1	2	2	0	0

** Note : Positive values indicate improvement and negative values indicate degradation*

Combination of Text and Design Metrics vs. Text Metrics

In Table 8.5, Δ(T-D) shows the classification performance comparisons between the text metrics and the design metrics as predictors. The results in Table 8.5 show the ability of classifiers to produce AUC scores ≥ 0.60 to all case studies. It shows that the performance of J48, k-NN(1), Decision Table, Decision Stump and Random Tree degrades while k-NN(5), Naive Bayes, RBF and Random Forests do not show any changes in performance. Only Logistic Regression shows an improvement where it could predict 2 more projects (compared to design metrics). We take a step further by combining the text metrics and the design metrics. We compare the performance of the combination of these metrics with the individual text metrics and design metrics performance. The comparison between the combination of design and text metrics (DT) and the text metrics (T) is shown in Δ(DT-T). Seven classification algorithms show improvement when using a combination of design and text metrics while k-NN(5) and Random Forests perform the same. Only logistic regression performance degrades but the difference is only one project.

Combination of Design and Text Metrics vs. Design Metrics

In Table 8.5, Δ(DT-D) shows that using a combination of text and design metrics performs better than using only design metrics. Six classification algorithms show improvements while three classification algorithms present the same result for both sets of predictors. However, k-NN(1) shows degradation in classification performance for Δ(T-D) and Δ(DT-D).

Random Forests in Detail

We analyse the results of the Random Forests algorithm in detail by comparing the prediction score for the three evaluated predictor sets (D, T and DT). In Table 8.6, five projects demonstrate that the text metrics produce better results than the design metrics. The improvement of text metrics using the Random Forests classifier is between 2% to 7%. However, another five projects show degradation. The degradation recorded are between -13% to -21%. On average, out of ten projects, the text metrics show degradation of 5.6% compared to the design metrics. From our observation, the text metrics perform better prediction than the object-oriented design metrics when the total InfoGain value of text metrics is higher than the total InfoGain value of design metrics (in other words, the text metrics dominate the influential features).

Table 8.6 presents that prediction performance by using a combination of both metrics is better than using only the design metrics. Seven out of ten projects show positive improvement. The improvement is between 1% to 22%. On average, by using the combination of both metrics leads to an improvement of 5.3% over just design metrics. Based on this data, it seems that the combination of the design and text metrics produces the best result for the prediction of class inclusion/exclusion.

Table 8.6: *Random Forests Result for Predictors Sets*

Project	Text (T)	Design (D)	Δ (T-D)	Improv. (%)	Design + Text (DT)	Δ (DT-D)	Improv. (%)
ArgoUML	0.672	0.655	0.017	2.60	0.799	0.144	21.98
gwt-Portlets	0.603	0.564	0.039	6.91	0.597	0.033	5.85
JavaClient	0.895	0.844	0.051	6.04	0.929	0.085	10.07
JGAP	0.788	0.748	0.040	5.35	0.794	0.046	6.15
JPMC	0.733	0.692	0.041	5.92	0.773	0.081	11.71
Mars Simulation	0.645	0.766	-0.121	-15.80	0.830	0.064	8.36
Maze	0.584	0.674	-0.090	-13.35	0.635	-0.039	-5.79
Neuroph	0.683	0.835	-0.152	-18.20	0.849	0.014	1.68
Wro4J	0.588	0.742	-0.154	-20.75	0.695	-0.047	-6.33
xUML	0.727	0.847	-0.120	-14.17	0.840	-0.007	-0.83
Average	**0.692**	**0.737**	**-0.045**	**-5.55**	**0.774**	**0.037**	**5.28**

8.7 Discussion

In this section, we summarize and explain the result in the previous section. We also describe the threat to validity for this study.

8.7.1 Text Metrics Predictors Performance

From the results in Section 8.6.1, we know that all individual text metrics are influential predictors. However, Table 8.4 shows that the text metrics are suitable for only four case studies (i.e. ArgoUML, JavaClient, Mars and xUML) The text metrics were not able to influence the prediction of the gwt-portlets. From our further analysis, the design metrics also were not influential in this case study. For this case study, the text metrics produce a better result where it can produce the AUC ≥ 0.60 (as shown in Table 8.7). This is not possible to be achieved using design metrics predictor. Probably, the combinations of the text metrics and design metrics for gwt-Portlets project influence the prediction. The InfoGain is suitable to estimate the individual predictive power. We could not see the influence of the combination of the text metrics and design metrics through InfoGain measure.

8.7.2 Classification Algorithms

By using the text metrics, we found that the Random Forests, k-NN and Logistic Regression are suitable to be used for prediction. However, by using a combination of the design and text metrics as predictors, we found that the k-NN, Naive Bayes, Decision Stump, Radial Basis Function (RBF) and Random Forests are suitable for this purpose. From these algorithms, the Random Forests classifier was chosen for the validation experiment because it produced the best AUC score among those classifiers. From the results, we found that the combination of the design and text metrics is the best set of predictors. Only k-NN (1) classifier showed a decrease in performance. The performance of the text metrics predictors for this classifier maybe influences this result.

8.7.3 Application of Classification Method

Based on the previous section, we have found that Random Forests is the most suitable classification algorithm for our approach. The Random Forests classifier is capable of producing scores for every class in a class diagram. Based on this, we developed a tool (called Software Architecture Abstractor (SAAbs) [124]) to apply our approach to the RE-CD (see Chapter 9). The tool produces a score of importance for all classes in a class diagram. A higher score indicates that the class is more important or could be included in class diagrams. With this ranking, abstractions of a class diagram based on the importance of classes can be constructed.

Table 8.7: *Classification Result from Text Predictor (T)*

Project	J48	k-NN (1)	k-NN (5)	Logi. Regr.	Naive Bayes	Dec. table	Dec. Stump	RBF	Rand. Forest	Rand. Tree	Project Performance
ArgoUML	0.54	0.59	0.66	0.77	0.80	0.53	0.75	0.76	0.69	0.56	**6**
std. dev	*0.08*	*0.06*	*0.08*	*0.04*	*0.02*	*0.09*	*0.06*	*0.02*	*0.07*	*0.04*	
gwt_portlet	0.52	0.60	0.61	0.65	0.63	0.50	0.57	0.53	0.63	0.56	**5**
std. dev	*0.05*	*0.07*	*0.10*	*0.13*	*0.10*	*0.00*	*0.06*	*0.10*	*0.12*	*0.10*	
JavaClient	0.84	0.85	0.90	0.86	0.82	0.80	0.72	0.86	0.90	0.82	**10**
std. dev	*0.04*	*0.04*	*0.03*	*0.05*	*0.03*	*0.06*	*0.04*	*0.04*	*0.04*	*0.07*	
JGAP	0.54	0.62	0.73	0.78	0.78	0.50	0.71	0.76	0.78	0.60	**8**
std. dev	*0.08*	*0.07*	*0.07*	*0.08*	*0.05*	*0.00*	*0.03*	*0.06*	*0.07*	*0.10*	
JPMC	0.74	0.61	0.66	0.70	0.64	0.52	0.72	0.68	0.73	0.61	**9**
std. dev	*0.05*	*0.07*	*0.07*	*0.07*	*0.07*	*0.06*	*0.02*	*0.11*	*0.12*	*0.09*	
Mars-Simulation	0.50	0.56	0.69	0.78	0.77	0.50	0.67	0.81	0.64	0.59	**6**
std. dev	*0.00*	*0.08*	*0.05*	*0.03*	*0.02*	*0.00*	*0.03*	*0.02*	*0.07*	*0.05*	
Maze	0.60	0.56	0.51	0.57	0.48	0.51	0.54	0.50	0.60	0.57	**2**
std. dev	*0.09*	*0.08*	*0.08*	*0.11*	*0.09*	*0.02*	*0.04*	*0.06*	*0.07*	*0.08*	
Neuroph	0.55	0.59	0.69	0.71	0.65	0.50	0.57	0.62	0.68	0.61	**6**
std. dev	*0.05*	*0.06*	*0.06*	*0.12*	*0.06*	*0.00*	*0.06*	*0.06*	*0.08*	*0.07*	
Wro4j	0.50	0.57	0.65	0.70	0.50	0.50	0.53	0.47	0.57	0.59	**2**
std. dev	*0.05*	*0.06*	*0.12*	*0.21*	*0.19*	*0.01*	*0.09*	*0.20*	*0.15*	*0.11*	
xUML	0.68	0.64	0.64	0.77	0.72	0.62	0.60	0.73	0.73	0.63	**10**
std. dev	*0.08*	*0.10*	*0.05*	*0.08*	*0.07*	*0.08*	*0.03*	*0.06*	*0.04*	*0.06*	
Algorithm Performance	**4**	**5**	**9**	**9**	**8**	**2**	**6**	**7**	**9**	**5**	

8.7.4 Threats to Validity

This subsection describes the threats to validity of this study.

Threats to Internal Validity. The splitting of class names is done by separating the words based on the occurrence of the capital letters. However, not all designs follow this naming convention. There are several uses of characters such as "XYZ" and "123" that do not carry any significant meaning. This leads to inaccuracy of counting the occurrences of words.

Threats to External Validity. It is difficult to find open source software systems that have both FD and source code. Hence, we use ten open source software systems in our evaluation. These systems fall into the class of small to medium size software projects. Hence, we could not generalize the result of this study of all software systems.

Threats to Construct Validity. To measure the classification algorithm performance, we use the AUC as our evaluation metrics. The AUC can be considered as a standard metrics in data mining [78] which is designed to evaluate imbalanced datasets [141]. Thus, we believe there is little threat to construct validity.

8.8 Conclusion and Future Work

This chapter aimed to come up with better methods for abstracting class diagrams out of source code. To this end, we focused on the use of class names as a way of improving the prediction of class inclusion/exclusion. In our study, we introduced text metrics that capture the frequency of occurrence of class names. We found text metrics may help to improve the results. However, using the text metrics in isolation does not produce better predictions. A combination of text metrics and design metrics (DT) produces the best result. We also studied which classifier algorithm works best for abstracting classes. Out of nine evaluated classification algorithms, all evaluated classification algorithms showed improvement in classifying class inclusion/exclusion by using the combination of text and design metrics except for the k-NN(1) algorithm. The evaluation was done by comparing the classification result with the result produced by using design metrics as predictors. The Random forest classifier is the most suitable for our study. By using this classification algorithm, we may improve the prediction on average by 5.3%; the improvement for different software projects ranges between -6% to 22%.

For future work, we plan to use additional sources of text in the source code (the methods, parameter and attribute names) to enhance the text metrics. We also conduct a user study on the usage of resultant diagrams constructed by the tool, which are developed using this approach (see Chapter 10).