A DEFECT PREDICTION MODEL FOR MODEL DRIVEN ENGINEERING

Kariyawasam Siththarage Dinesh Nadun Kumara de Silva

168214E

MSc in Computer Science specializing in Software Architecture

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

May 2020

DECLARATION

I declare that this is my own work and this Post Graduate Degree Project Report does not incorporate without acknowledgment any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief, it does not contain any material previously published or written by another person except where the acknowledgment is made in the text.

Also, I hereby grant to the University of Moratuwa the non-exclusive right to reproduce and distribute my thesis, in whole or in part in print, electronic, or another medium. I retain the right to use this content in whole or part in future works.

05/29/2020

Kariyawasam Siththarage Dinesh Nadun Kumara de Silva Date

I certify that the declaration above by the candidate is true to the best of my knowledge and that this project report is acceptable for evaluation for the MSc Research.

.....
.....

Dr. Dulani Meedeniya Date

Acknowledgment

My sincere appreciation I need to gift to my family for the continuous support and motivation given to make this thesis a success. I also express my heartfelt gratitude to Dr. Indika Perera and Dr. Dulani Meedeniya, for the supervision and advice given throughout to make this research a success. All my MSc batch mates who contribute with myself throughout the study of this entire duration in my stay in the university for this Post Graduate Study has to be appreciated as well. My parents and my relatives have to be thankful for understanding my busy schedule and the fact that I did not have enough time to spend with them and become a close family member or a relative. Last but not least my newborn child Vihini and the loving wife Lakmini for making me smile throughout this tough period.

Abstract

Model-Driven Engineering (MDE) is used in the Software Industry which enables level to level transformation until the final system is created. This concept helps to ensure the bridging of gaps between the problem domain and the solution scope of a software system. A software system with a lesser number of software defects or zero defects will be successful software. Earlier the defects are identified it reduces the cost in terms of effort, time, and human resources, rather fixes that defect in a later stage of the software development life cycle. The development of defect prediction models and the efficient usage will prevent unnecessary defect fixing efforts later.

Unified Modelling Language (UML) provides certain notations to create models in different aspects. UML Class diagram is very widely used in identifying and evolving business entities. UML Class diagrams as entities can be mapped with Database Management Systems and propagate the business entities.

Every business must deal with the inevitable truth of change. To survive in a competitive market, business functions, and business directions are under the freedom of change at any moment. Stable business solutions are the compulsory components of successful businesses.

Applying changes to Software makes them fragile when they are not done properly. The phase where adding changes or in the maintenance mode, and the most important stage of the business, must be well away from defects.

This thesis covers a defect predictive approach that can be applied at the UML class diagram models that are created at the beginning of the Software solution, however, the defect prevention applies to the maintenance or in the most vital stage of the business.

This thesis discusses the possible defect prediction models that can be used in MDE to facilitate fast and efficient software development. At the end of the thesis, it will discuss the approach that has taken to introduce a defect prediction strategy to the Model-Driven Engineering its evaluation and the contributions to the research community

Table of Contents

Table of Figures

List of Tables

List of abbreviations

EDOC	Enterprise Distributed Object Computing
EJB	Enterprise Java Beans
ILLE-SVM	Improved Local Linear Embedding and Support Vector Machines
IOT	Internet of Things
LLE-SVM	Local Linear Embedding and Support Vector Machines
MDA	Model Driven Architecture
MDP	Metrics Data Program
MOF	Meta Object Facility
NN	Neural Networks
OMG	Object Management Group
PIM	Platform Independent Models
PSM	Platform Specific Models
RF	Random Forest
SDLC	Software Development Life Cycle
SNB	Supervised Naïve Bayes
SVR	Support Vector Regression
UML	Unified Modelling Language

List of Appendices

Chapter 01

Introduction

1.1 Background

1.1.1 What is a Model?

There are certain ways that the human brain can work on problems and solution deriving. The human mind tends to work continuously on real problems repeatedly by applying cognitive processes [1]. In that process Abstraction is an important action. The ability of finding the commonalities in many different observations. Reality or the problem can be mapped into the human mind and can be expressed via Models as a partial or simplified version. The expression done through the Models can be used as the inputs of processing the next abstraction levels of the problem solving and expressing it to an audience.

In the Software Engineering domain, Models can be considered as the central artifact of Software Development. Models can be used as the artifacts for the following sections [3].

1. Prototyping
2. Code Generation
3. Automated Testing
4. Refactoring or Transformation
5. Documentation
6. Analyzing

1.1.2 Model-Driven Engineering

There are several problems associated with developing successful defect-free complex software. Software development technologies have evolved to cater to the complexity of software development and deliver defect-free software. Agile Manifesto [4] is a customer-centric approach that focuses on customer satisfaction as the focus of software development. Extreme programming practices are encouraged to follow and ensure the coding standards are followed to make the code is a bug-free, readable, and maintainable one. Any kind of software development needs to involve with as a significant factor is a gap or difference between a problem domain and its solution or implementation domain (Problem Implementation Gap) [2]. Correct bridging of this gap almost ensures a successful software with a lower number of defects.

Model-Driven Engineering (MDE) researches focus on understanding and reducing the gap or distance between the problem domain and the solution implementations. The hardness or the complexity of reducing this difference is handled with the technical support to transform the problem abstractions to solution implementations. In MDE the complexity is explained with models in multiple levels of abstractions and with different variety of perspectives. Model transformation happens between the different abstraction levels. From the MDE viewpoint in software development, Models play the role of being the primary artifacts of software development. Models that are used in the runtime environment and its behavior also another research topic in MDE. All the researches belong to the MDE focuses on the methodologies that can be used to minimize the complexity of the software and the implementation platform [1].

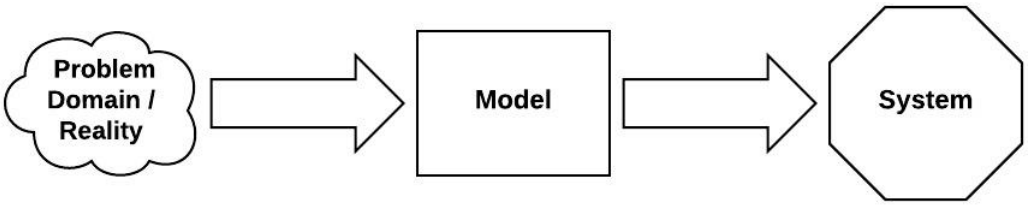

Figure 1.1: Modelling in between reality and the expectation

There are 3 types of modeling levels that can be used based on the abstraction levels [1].

- Computational Independent Models (CIM)
- Platform Independent Models (PIM)
- Platform Specific Models (PSM)

CIM are the models of very high abstract level that explain the requirements and other needs. User requirements, business objectives can be embedded into these models.

PIMs used to explain the behavior of the systems in terms of stored data and behavioral algorithms without any technical or technological details.

PSMs used to define and explain the technological points of views in to the relevant Models.

There are other factors behind using Models for Software Engineering. Software becomes more and more complex. Customer expectations from the User Interfaces, various evolutionary viewpoints like security, performance, and network capabilities are few explanations of how the Software can be more complex even in a shorter period. There are various types of devices that the Software is used. Now it is not like the era of having just small pieces of Software for the simple devices supposed to work a simple task like television or the washing machine. Emerging various types of devices and the modern concepts of interconnecting the devices with the concept of the Internet of Things (IoT) allows the Software to play a vital role and make them complex.

Software Defect or a bug is a deviation of the expected behavior or a requirement of a system. Lesser number of defects in a software system is an indication of a reliable software system. The cost of fixing a defect in a later stage of the software development life cycle is much higher than an earlier stage of it. The earliest defect identification is a worth investment for a system. Defect prediction is identifying possible defect prone areas of a system and make sure defects are avoided or reduced. Defect prediction models guide to identify the defects of a software system.

In MDE identification of a defect in the abstraction levels closer to the problem domain prevents propagating those defects appearing in the later abstraction levels and the models. Another defect prediction model that can identify the defects in the models closer to the solution implementation also can be the improvements for the models for the higher-level models. A possible prediction for defects in MDE is a solution to a problem that is in a lower level of abstraction than those used to express the problem domain. Improper problem solution bridging is a root cause of defects in MDE.

1.1.3 MDE Approaches and challenges

Model-Driven Architecture (MDA) a larger initiative on MDE by the Object Management Group (OMG). The main goals of this initiative are interoperability , portability and reusability achieved by architectural separation of concerns. MDA is

not a new methodology introduced by OMG, however an approach for software development with the existing OMG specifications. Existing OMG specifications are

1. Unified Modelling Language or UML
2. Meta Object Facility or MOF
3. Common Warehouse Met model or CWM.

There are other practised technologies as follows as well.

1. Enterprise Distributed Object Computing or EDOC with its mapping to EJB
2. CORBA Component Model or CCM.

In MDA, Model is defined as a recognized representation of a structure, function and the given context system behavior and a specified viewpoint. The approach based on model, used as the basic resource for the following purposes of a system.

1. Documenting
2. Analyzing
3. Designing
4. Constructing
5. Deploying
6. Maintaining

All together MDA is the combination of the specification or the documentation of the connectors and parts of the system. MDA allows rules for the interactions of the parts using the connectors.

Bridging the gap between problem and implementation is the basic research concern in MDE. There is a perception that the models are the primary documenting artifacts and then they are rarely used in later software development. These kinds of limitations need to be properly addressed when starting to get the full advantages of MDE. Growing complexity of software systems and different platforms and variations of software development strategies can be widening the problem implementation gap.

Challenges associated with MDE can be briefly categorized as following.

- Modelling Language Challenges – The language that use to model and its constraints may limit the possibility to express problem level abstractions.
- Separation of concerns challenges – Problems associated in modelling in multiple viewpoints, overlapping viewpoints
- Model manipulation and management challenges – The possible challenges linked with analyzing, defining, with the model transformations, traceability maintaining linking to the model evolution, and also backward engineering, maintaining consistency among several viewpoints, tracking versions, and runtime model usage.

1.1.4 Defect Prediction

In Software Engineering, Defects are supposed as the deviations of the expected behavior of a system. The earlier a defect is uncovered; it is saving the cost in terms of time and money rather a defect is covered in a final step of a Software Development Life Cycle or commonly referred as SDLC. It is much important to have Defect prediction in the first stages of the SDLC. Software Defect prediction had researches on various aspects due the importance of it.

1.2 Research Problem

Model-Driven Engineering is an approach that is used in Software Development. Due to the high cost of defects solving after the solutions are in the production environment, It is highly desirable to prevent defects in Software development to reduce effort on fixing those. Another important aspect of Software Development is making the software ready to absorb the changes. Absorbing changes happen in the maintenance phase of ongoing business, which raises a high probability of defect arising with the highest cost of solving issues. Model-Driven Engineering based Software Systems do not assure completely defect-free systems. Therefore, the Software defects that can be appeared in the Model-Driven Engineering is considered as the research problem.

Research problems are scoped to the UML diagrams that are used to model class diagrams. Building class diagrams is a primary design aspect used in Object-Oriented Design. It requires identifying the business requirements, analyzes them,

understands the stakeholders, and then identify the business entities. Those business entities can start to model with the UML class diagram.

In modern-day the challenge is to stay stable with the required changes and the requirement additions into the system. A bad design will make the solution more fragile gradually along with the required changes and additions. A solution becomes fragile means it throws defects that interrupts the functional and nonfunctional expectations. Knowing the fact that it is unavoidable that the required changes and additions to the design if we can get ready to those at the very beginning of the system design, it will be a healthy defect prediction approach.

1.2.1 Research Problem Statement

When software is implemented using models in the implementation phase, when it comes to the maintenance phase the existing code implementation tends to raise defects. This problem identified as the research problem and a Defect prediction methodology that predicts defects in MDE is proposed as the solution.

This Defect prediction methodology that predicts defects in UML Class Diagrams. Understanding and solving UML Class Diagram model level design problems will prevent defects are propagated into the source code.

1.3 Proposed Solution

In this research, there is a proposal for a Defect Prediction methodology and applies to the MDE and reduces the possibility of arising defects and predicting the defects early in the SDLC while using MDE.

In this solution it will identify the Object-Oriented Design Principles as the Defect preventing methodologies and emphasize the value of it. Then it will come up with a solution of evaluating UML Class diagram designs evaluating the Design Principles against the custom designs.

1.4 Research Objective

The objectives of the research are as follows.

- Identify the Model-Driven Engineering approaches and its importance
- Find out defect prediction methodologies and models that are used.

- Identify the probability of Software Defects that can be raised in the maintenance phase of the Software Development Life Cycle.
- Explain the proposed Defect Prediction Methodology.

1.5 Research Overview

In this research, it is intended to find out a defect prediction methodology for Model-Driven Engineering. Model-Driven Engineering is explained, and the concept of defect prediction models also will get investigated. Due to the high importance of reducing defects in the software maintenance phase of the Software Development process, defect prediction methodology is targeted at the maintenance mode of a Software.

Chapter 02

Literature Review

2.1 Model Driven Engineering

There are several Model-Driven Engineering based Software implementations done over the past.

There was a research study[52] done towards automated view abstraction for distributed model-driven service development. UML is widely used in the area of Model-Driven Development and the Model transformation. When the processing is happening among the models, empirical processing rules are applied to model transformations, model simplifications, and check consistency and reducing complexity. However these empirical rules itself have some challenges associated like validating completeness of the model, consistency can be existing within the rules, and managing priority in the composition. As the solution proposed, constructing finite-state automation is supposed to perform the activities done by the empirical abstraction rules. The following notations are used to represent UML activities.

Table 2.1 Shorthands for UML Relationships

Relationship	Shorthand
Generalization	GL
Dependency	DP
Association	AS
Composition	CP
Aggregation	AG
…Reverse	…

The following diagram is an example of how different classes are interconnected. Class Human is connected to other Classes, Car, Company, Animals, Plants, Men, Water, Bird.

Using the above notations, the following example model has been analyzed.

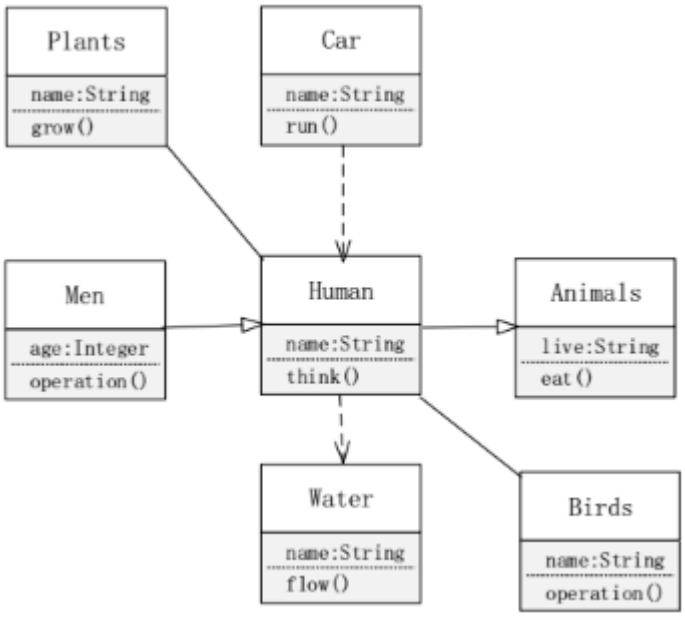

Figure 2.1: Example Model [3]

The following notations are derived from the example model.

Triple relations that start from the Class named Men are as follows.

(Men) - GL- (Human) – GL - (Animals)

(Men) – GL - (Human) – AS - (Plants)

The triple relations which start from Class named Plant as follows.

(Plant) – AS - (Human) – GL - (Animals)

(Plant) – AS - (Human) – GL - (Men)

Finite automation has been applied afterward and used that methodology for automated view abstraction.

There is another research is discussed under this chapter that is Model-Driven Development of Web applications using AngularJS Framework[15]. Today's competitive business environment forces Software Companies to release client requirements through software products, software changes as quickly as possible, and wants to deliver such products in lesser time while keeping the quality of the products high. Model-Driven Architecture (MDA) introduced by Object

Management Group (OMG) is facilitating this. MDA allows transforming a model of software to another one.

Models in the software can be transformed into Platform Independent (PIM) Model to Platform Specific Model. PIMs represent a higher level of Models which represents the business functionalities that are independent of the platform that the solution is implemented and the code level that the solution is implemented. PSMs are the next level of Models that are coupled to the platform that the solution is implemented. In that way we can build a hierarchy of models that can be transformed from one level to another. This approach reduces the time to implement the solutions and reduces the possibility of introducing bugs.

In this use case [15] a web application is considered. To make the web application in a high performing one it is built following the Single Page Application approach. When using the Single Page approach, most of the code is essential to code is loaded in one single load and stored on the client-side. Anything that not available with the loaded content will be fetched dynamically on demand. Google's AngularJS is used to achieve the Single Page Application concept here. AngularJS design for rendering dynamic views that allows the browser page components rendering happens smoothly just like similar in a native application.

PIMs can be transformed into PSMs with the transformation tools by following certain transformation rules. PSMs are used in this case study and development done based on that. UML Profile [16] mechanism has been used along with the UML Models with the extending of UML Models. In this case, PIMs are converted to PSMs using UML Profile. UML Profile supports Stereotypes denoted by << >>, and Stereo values and constraints.

UML Profile of this case study is illustrated as follows. The stereotype AngularApp explains a web application in Angular in this profile. Using AppRouter, a new view can be added into the angular application. This view is represented by the stereotype AppView. This view will be loaded into the web application without getting the whole HTML content from the web server.

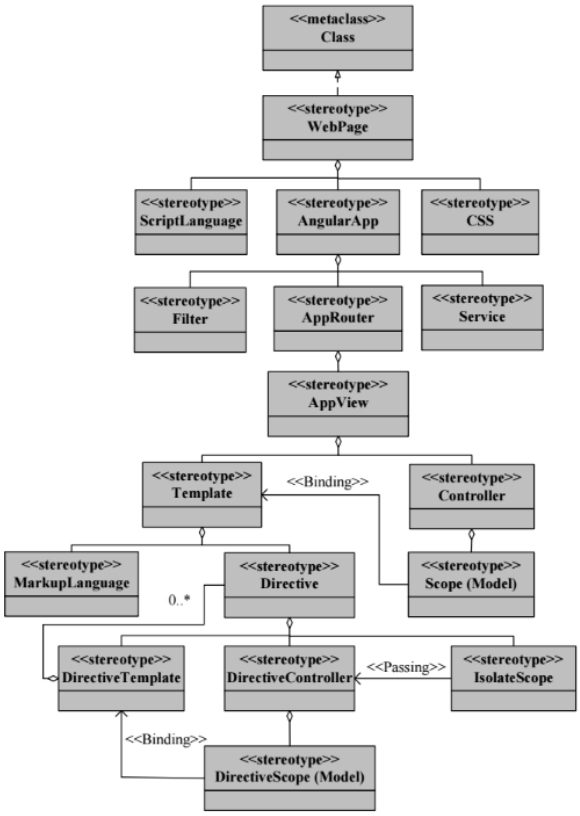

Figure 2.2: Overview of UML Profile for this case study [15]

Model-View-Controller (MVC) pattern can be observed in the Directive featuring DerectiveTemplate as the view, DirectiveScope as the model, and DirectiveController as the controller. IsolateScope can be described as a property that has a value assigned from external or away from the Directive. The value of IsolateScope is transferred to the DirectiveController for the process of particular business logic.

For simplicity, the model to model transformation is shown as the form of AngularJS code templates. These templates are addressed again in the case study.

Every component showed in the above diagram is explained in the following table.

Table 2.2: Stereotypes for this case study[15]

Stereotype	UML Metaclass	Description
AngularApp	Class	Define the while AngularJS application
AppRouter	Class	Define which view to be navigated to
AppView	Class	Define the whole user interface of the single web page
Binding	Association	Represent the relationship between Scope and Template. If Scope is changed, it will immediately affect Template that is showing the Scope value on the web browser
Controller	Artifact	Function to precess all business logic
CSS	Artifact	CSS tag on the web application
Directive	Artifact	AngularJS directive
Directive Template	Artifact	Partial HTML of Directive
Directive Controller	Artifact	Function to process all business logic inside a directive
Directive Scope	Artifact	Model to be used on Directive
Filter	Artifact	Function to format any value of AngularJS application
IsolateScope	Artifact	Model to be used inside the directive
Markup Language	Class	HTML tag to render on the client-side
Passing	Association	Association between IsolateScope and DirectiveController. IsolateScope will path the value that is set from the Directive to DirectiveController to process any business logic.
Scope	Artifact	Model to be used on the template
ScriptLanguage	Class	JavaScript file to be used in the web application
Service	Artifact	Shared functions to process specific business logic across the AngularJS application.
Template	Artifact	The user interface of AnualrJS view
WebPage	Class	The webpage that is sent from the webserver.

PSM Models of the above stereotypes are as follows.

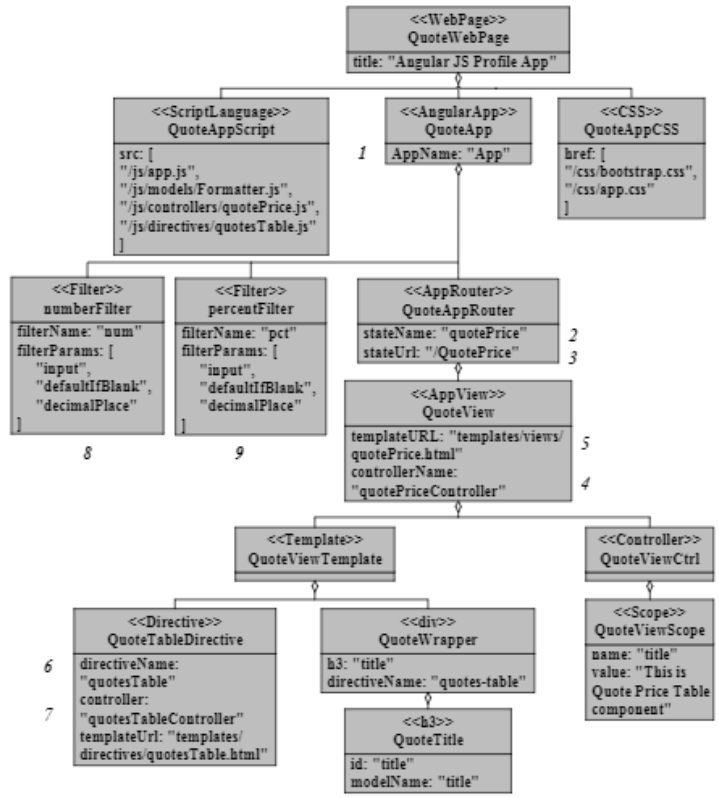

Figure 2.3: Design of the model in the case study on UML Profile - Angular JS [15]

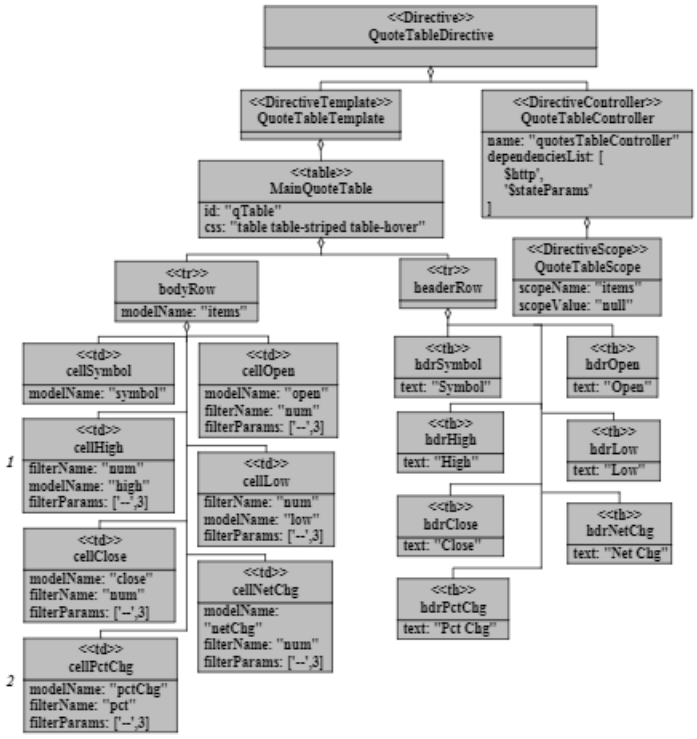

Figure 2.4: Directive section – case study design model [15]

Using this approach, they could generate most of the code and its adjustments only need to be done.

As a modelling language, Unified Modelling Language (UML) is widely used. To make UML diagrams more informative, Object Constraint Language is used (OCL). It helps to transform models to code. In this scenario before converting the model into the code, it is required to find the defects can be associated with the model. It prevents spreading defects in the model move into code and reduce effort to correct both model and the code. There are existing tools that can use to verify the accuracy of UML and OCL models. But an identified limitation of such tools is their computational complexity that limits the scalability. Those formal tools must make large computational activities to evaluate a class diagram. So, when a class diagram scales larger gradually, that computational complexity increases a lot. Verification execution time depend upon the classes and its count, their association relations, and the constraints noted by OCL they have. So the formal tools face the challenge of completing the verification within a given time frame due to mentioned constraints.

As a solution, model slicing techniques have been introduced [19] called the UML/OCL slicking technique (UOST) as follows.

1. Slicing or dividing of a disjoint set of sub-models.
2. Slicing or dividing of a non-disjoint set of sub-models.

The dividing or slicing technique here divides a model p into sub-models like p_1, p_2, p_3….p_n. In this scenario it needs to satisfy p_1, p_2, p_3, …. p_n all sub-models to satisfy model p. When you take the disjoint slicing, it does not help to create sub-models in every scenario. For example when there is a common class that combines with OCL constraints, the sub-model p1 equals the input model p. In such a scenario disjoint model does not provide any sort of advantage to the verification process. However in no-disjoint slicing, it goes beyond that limitation and further allows to the creation of sub-models. If a model has classes cl1, cl2 and cl3 and it has OCL constraints c1 and c2, the model M = {cl1, cl2, cl3} constraints C = {c1, c2}. If c1 constraint controls the classes cl1, cl2 and c2 controls the classes cl2 and cl3, the non-disjoint slicing can apply to the model and create sub-model s1 = {cl1, cl2} and s2 = {cl2, cl3}. In disjoint slicing, s1 = {cl1, cl2, cl3}. They also provide a feedback technique to identify un-satisfiable sub-models. Then the developers can save time by targeting

the un-satisfiable models and correcting them. In this paper they derive the following 3 hypotheses.

1. By applying slicing techniques, the verification time could be reduced.
2. More complex UML/OCL diagram verification will enable by slicing.
3. Without considering the formalism of the existing verification tools, the implementation of the slicing technique should be possible.

In their research method, they have followed the design and qualitative research methods. They have raised a questionnaire of 13 questions to 80 researches, software designers, and domain experts. The purpose was to confirm the problems and the scalability of the complex UML/OCL model problems are aligning with the research direction. In the quantitative approach they have taken, the methodology that improves the verification process is focused.

Their work has motivated by the tool UMLtoCSP [20] that is used in formal verifications of the UML class diagram. It suffers from the scalability problem with the class diagram getting more complex. They have followed an in detailed description [21] of class diagram verification using constraint programming. This team renamed the tool UMLtoCSP into UMLtoCSP (UOST) after implementing disjoint slicing, known - disjoint slicing, and feedback technique algorithms.

When OCL constraints added class diagram is given to the UMLtoCSP tool, the accuracy of the properties is checked automatically. It can evaluate that the measured properties are either strongly satisfiable or weakly satisfiable. Background techniques of the UMLtoCSP tool are explained in Constraint Logic Programming [20]. For the verification engine, it uses the ECLiPSe [21] constraint solver. Object Constraint Language specification is introduced by Object Management Group [22] and they created parts of UML standards [23].

2.2 Defect Prediction

There are several types of research conducted on defect prediction. The different contexts are used when developing different Defect Predictions. The following explanation tries to explain those defect prediction methodologies in different categories.

2.2.1 Data Mining and Machine Learning

Recent research[12] shows that how the Support Vector Regression – SVR can have a considerable possibility on Software Defect Prediction. That research was about predicting Software defects depending on Local Linear Embedding and Support Vector Machines algorithm or LLE-SVM. However using the Grid search algorithm, parameter efficiency usage of the LLE-SVM model is expensive in computational power usage. This drives to the lower efficiency of the model. A solution proposed for this problem based on a model used for predicting defects depend on improved LLE-SVM is proposed. Depending on an Improved Locally Linear Embedding and Support Vector Machines - ILLE-SVM this solution is approached. This model uses a search algorithm called a coarse-to-fine grid, to find the most optimal parameters. This model ensures better correctness of the parameters a reduced parameter enhancing time by step by step narrowing of the scope of search and parameter step. LLE-SVM and ILLE-SVE are compared on four NASA defect datasets and proved that than the LLE-SVM, ILLE-SVM can search the optimal parameters.

Another Machine Learning and Data Mining based research have conducted which is Superposed Naïve Bayes for Accurate and Interpretable Prediction [13]. They were targeting a model of prediction which high accuracy and with a higher power of explanatory. They highlighted the following research questions.

- Can the Naïve Bayes ensemble process achieve higher performance than any other standard classification technique?
- Can Naïve Bayes ensemble be transformed into a simple Naïve Bayes model without degrading the performance?

To address the above research questions, they come up with a new algorithm using Naïve Bayes classifier called Supervised Naïve Bayes (SNB). It first builds the Naïve Bayes ensemble and then it transforms it into Simple Naïve Bayes Model. NASA (MDP) Metrics Data Program datasets are used to evaluate the results.

Considering the potential of using Data Science, this research is based on using 3 machine language algorithms

1. Random Forest - RF
2. Neural Networks - NN

3. Support Vector Machine - SVM [6].

This research contains the usages of these algorithms that are used to predict software defects. Using those algorithms, they tried to figure out the most defect probable components of the files in a particular software project.

Semi-supervised learning, with change burst information, is used in research that tries to address two problems related to Software Defect Prediction [5].

Since most of the defect predictions take project data history versions and to predict potential defects, early discovered defects are again referred. Anyway, in the early stage of a project, there is a lack of project data and the defects. Therefore, there is a problem of predicting defects in the early stages of the Software Development Life Cycle.

Using static code matrices as the predictors add risk to the defect prediction since it does not consider the information change that can add several risk types into software development.

After taking consideration above problems, Semi Supervised approach has been proposed for Defect Prediction ability. With the support of classical supervised Random Forest algorithm using a self-training paradigm, extRF is extended. Further, it deploys change burst information in order to obtain the improved accuracy of software defect prediction.

Learning and simulating the association between the metric specifying the object orientation and the concept of change proneness [7]. Following two views has been addressed,

- Quality, functionality, and productivity that affection from Parameter qualification.
- Usage of Machine learning techniques for predicting software.

Using Machine Learning for Software Defect Prediction is further investigated and collected data manipulation and defect prediction is done with a well-known WEKA tool that uses for data mining activities and the visualizations.

2.2.2 Security Defect Prediction

Defect prediction has extended to the various types of viewpoints. From the security point of view, the possibility of emerging security issues is considered as well. Analyzing and assessing the security-based defects has been done [8] and the outputs are considered focusing on cybersecurity issues. Such defect prediction models can be used to predict various types of aspects of security-related defects.

There was another study [14] done on the security analysis of the system built. It has identified different data from five projects either as safety-critical or security-critical. Some other parameters in Software implementation that contribute directly or indirectly towards the security of the system, like budget, cost, schedules, etc. This research has identified a correlation between modeling quality and security.

2.2.3 Other Defect Prediction approaches

There is a considerable potential on the source code itself to predict issues. Research [9] shows that a code that can handle exceptions has a possibility of being a riskier code than a code that does not handle exceptions. It introduces a specific framework to predict software faults from annotated exception handling method call structures which is explained in the extended abstract. This framework generates annotated exception call graphs of the whole system and calculates property-based software engineering measurement values.

There is a term ‘Exception defect’ [47] is described as an improper implementation of the exception handling code. To analyze the behavior of the source code, call graph [48] is used and it will show how the relationship between methods in nodes. In the same way a call graph can be generated to the exception handling path. Not only this research study but also previous investigations [49] highlights an interesting fact that a code that take care of exception handling, can have more defects than a code that do not pay much attention on exception handling. Further those facts highlight that the defects that are in the exception handling code, more resides in the exception handling code flow. Call graphs in exception flow are further wrapped with the properties associated with the flow. On the software measurements [50] carried on property-based measurements, how the properties are used is further explained.

The research problem they have isolated as the structural attributes in the call graphs of exception handling, that are measured with the property depended software engineering measurements, can be used to predict the defect prone possibilities in a system or not.

The solution in the research, they have identified the independent variables and the dependent variable used along with multivariate statistical models and univariate statistical models. Dependent variables they used are the length, size, coupling, complexity, and cohesion of the exceptions call graph structures. The dependent variable is a module is defect prone or not. One or more bug having modules here is considered as bug prone.

Further to identify the relation between the exception call structures and the defects, case studies are performed, and an experiment performed with eight processes [51].

1. Data Collection

   In this phase, the following data are collected.

   a. Properties of cohesion of exception call graphs, coupling, length, size, and complexity

   b. Class and method list of exception call graphs

   c. List of bugs in all classes used in the case study

2. Data pre-processing

   a. Process of understanding the distribution of collected data. Data is validated whether they can be used or not.

3. Dataset creation

   a. In this step fit data and test data set created for each use case study.

4. Principle component analysis

   a. The optional process used here. This is a method for converting independent variables into an orthogonal set of variables.

5. Formulation

a. Developing a formula for the prediction model happens in this step. The characteristics of independent and dependent variables are considered here.

$$\log(\frac{p}{1-p}) = \beta_0 + \beta_1 x_1 + ... + \beta_j x_j + \beta_m x_m$$

p is the fault probability. β is regression co-efficient. m is the number of independent variable and

6. Calibration

    a. This is the step where the relationship pattern between independent variables and the dependent variables are setup.

7. Prediction

    a. With the model obtained in Calibration, results are predicted.

8. Evaluation

    a. In this step received predicted values are assessed.

Eclipse code bases of version 3.1 to 3.6 are measured here since Eclipse releases pass their initial criteria.

The following chart shows the comparison of with and without exception handling constructs deviation against the size of the classes.

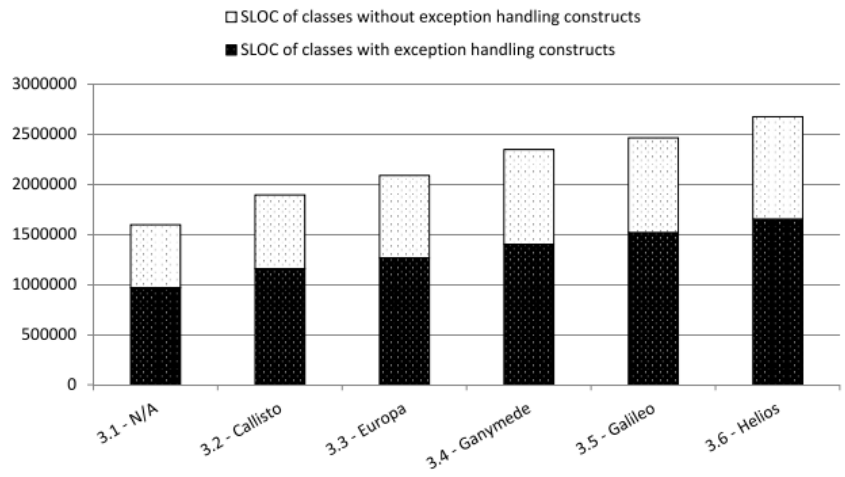

Figure 2.5: With and without Exception handling constructs against the class size

The comparison between the overall defects and the exception class defects is expressed in the following diagram.

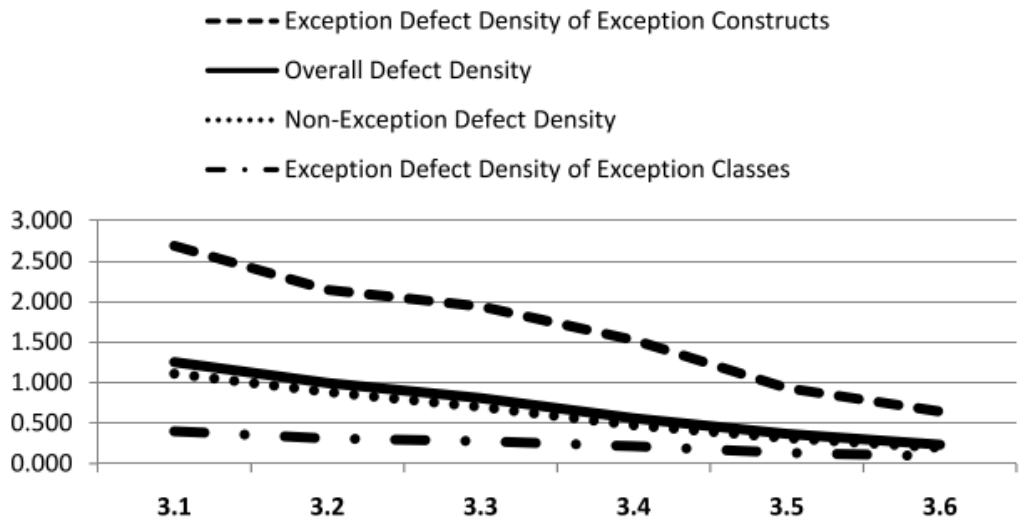

Figure 2.6: Comparison between overall defects and exception class defects

Another research[10] carried out in order to identify Software matrices on Defect prediction was when a model that can predict defects is trained, the defect data in the history often taken from a source code repository or a Subversion like version controlling system. In the file level Software change entries can be kept. However, the software matrices are obtained at the level of Class or the level of method. To bridge the file level and Software metrics level disagreement, most of the time Software related Matrices are combined to file system level most probably using summation. This research studied how to use different aggregation methods or schemes that can be impacted by Defect Prediction Models. They went through data retrieved from 255 open-source projects given 11 aggregation schemes. Two specific findings of this study are

1. When the correlations are considered in between software matrices and also the correlations among software matrices and the count of the defects, aggregation schemes can significantly change those correlations

2. When building models to forecast the defect possibility or its potential, which aggregation scheme to be used is examined. When summation only used, 11% of studied projects give the best performance while aggregated schemes are showing 40% of studied cases at its best performance.

3. When models are built to forecast defect count or rank, it gives similar or closer best performance using the only summation and manipulating all of the investigated aggregation schemes.
4. When models are built for defect prediction with effort awareness, aggregation schemes with mean or median results, that are close to the highest performance than the other aggregated scheme in any means.

In a broader view, this research team goes beyond their community's practice of using only a summation aggregation scheme to find the possible potential of other schemes. They further advise the reading audience to try furthermore aggregation schemes.

There is a survey [11] conducted on Topic Models which are predicted to be efficient to mine not structured data in Software Engineering. They have analyzed the various Software Engineering tasks that have taken place between 2003 and 2015. Among the Software Engineering tasks, trace-ability link recovery, source code comprehension, Software testing, developer recommendation, refactoring, Software history comprehension, social software engineering and Software Defect Prediction. Related to this research scope, findings related to Defect prediction are applicable.

In this paper they refer to one another source [43] that uses machine learning for software defects prediction models. A topic model to measure problems of the code used as an input to the machine learning-based approach.

A model named Delta Latent Dirichlet Allocation model is implemented in another source [44]. Failed program run execution traces are analyzed. There are latent topics two types have considered here. Those are Bug topics and Normal usage topics.

Another source discussed is statistics-based topics modeling [45], which is used to evaluate software problems as topics. Different various topics are used to help explain the source code's defect probability.

Another source [46] indicates another approach taken to predict source code defects. Relation strength is measured by a model named citation influence topic model. This analyses the dependency strengths between source code and developers.

Selecting parameters or the matrics to predict the defects is important. Research is done with the main objective of identifying the matrices for a defect prediction model [24]. Modern Software large in nature and has many different aspects as well. With the usage of wrapper and filter techniques, this research proposes a two-novel hybrid Software defect prediction model. This method collects metric selection and training processes into one process and it allows the reduction of the measurement overhead significantly. Quantitative measurement of fixing the cost of a defect uncovered after the production deployment is mentioned as 100 times higher than fixing it during the development phase [25]. In the same way, prediction defects in early stages are hardly possible due to the unavailability of the possible failures at those stages [26]. The matrices that use to measure software reliability can be internal or external. The source code level measurements are supposed to be internal matrices that the designers, and developers involved in major. Therefore, the developers need to pick the best internal matrices that can cause defects in the early stages of the cycle, and later those can make the key matrices for monitoring. Also, developers need to be aware of the changes in product, process, and resources as different versions are released. The examples maybe

1. The source code design
2. The source code itself
3. Regression-based tests
4. Testing requirements
5. Resources and Processes in between releases.

However, in practical concerns, it is hard and expensive for a developer to track all the matrices and monitor, measure, analyze them.

For the automated prediction of the internal matrices standard techniques are proposed along with the defect data from similar projects.

1. Naïve Bayes [27]
2. Support Vector Machines [28]
3. Decision Trees [29]

4. Neural Networks [30]

The approaches that use to select the matrices are as follows.

1. Filter approach [31]
2. Wrapper approach [32]

A hybrid model is discussed in this research team from the above two as a wrapper-filter method for software defect prediction and appropiate metric selection. The difference in this approach is hybridized with filter approaches and different wrapper heuristics. However the Wrapper heuristic-based hybrid models were introduced, SVM hybrid heuristics and Artificial Neural Network hybrid heuristics. Then those hybrids applied to find the target of this research, the matrices that could be used to predict defects from software defect data.

In brief they

1. Create a single optimization problem with defect forecasting problem and main metrics selecting the problem
2. Introduce a framework for identifying significant matrices, with a mix up of different wrapper techniques and the filter techniques. Within this step it combines the training process and metric selection of defect prediction into a single process.
3. Coupling SVM and Artificial Neural Network, a hybrid model filter created for higher prediction accuracies.
4. The two wrappers entities and relevant wrapper values or scores used along with the filter score to justify the performance of hybrid algorithms.

Different researches have used several matrices for Software defect production. Such research [33] has used the following aspects for the defect prediction.

1. Object coupling each other
2. Class response from other class.
3. Coupling based on Message passing.
4. Coupling based on Information flow

Another systematic review [34] based on Software defect prediction mentioned that most correlated OO matrices are Coupling between Objects, Response for a class, and Lines of Code.

However in this research, they have considered following matrices

1. Lines of Code
    a. A very basic metric. Includes executable code lines, blank or empty lines, and not executing codes. There is a lot of researches [35] that derives a rough relation between lines of code and defects count in the software.
2. The MCCABE Cyclomatic Complexity (MCC) metric.
    a. This calculates the logic of independence through the source code of a program. It is a representation of the complexity of the source code which is about to test. This metric is proved [36] as an important metric for software defect prediction.
    b. This metric can be further explained. Graph G is explaining the control flow of a software product. Individual nodes are relevant to a piece of sequential code and each edge relevant to a path created by a decision. $V(G) = e – n + 2p$, e is the what is the number of edges in that graph, n is the count of nodes in the graph, and p is the number of connected modules or components in the graph.
3. The MCCABE Essential Complexity Metric (MEC)
    a. The amount of bad logic structuring in source code is measured by this metric. The control sequence structures like if-else, for loops, while loops, etc., into a single exit point that helps to measure how good a software program is structured. When a high MCC value is showing that it is hard to test, and if that same code can be converted into the show a low MEC which means the program has broken into smaller functions, those functions can be tested easily.
4. The MCCABE Module Design Complexity Metric (MMDC)

a. When a subroutine or a flow is called by the module, this metric measures the number of decision logics involved in that flow. A detailed analysis [40] recommends that this MMDC metric should be used in software defect prediction. How this metric is calculated is by removing the nodes that represent the decision logic in a way that does not impact the calling flow of modules over its subroutines. As an example in Figure 2.7 nodes 8 to node 12 could be reduced to a single node that does not have any impact on the calling control of the module over the subroutines in sub () and subB().

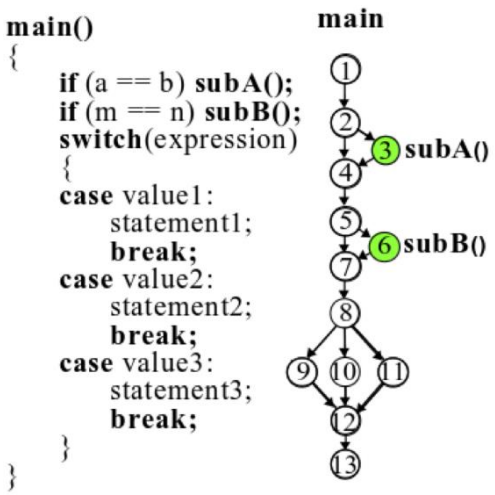

Figure 2.7: A sample program source code with control flow

5. Halstead Metrics

a. This evaluates some primary parameters in the software program which are its development time, length, effort, and other factors. The primary parameters as takes as follows.

i. n_1 = the number of distinct operations

ii. n_2 = the distinct amount of operands

iii. N_1 = the sum of operators

iv. N_2 = the sum of operands

Several measurements will be manipulated base on the above parameters as follows.

- Program Vocabulary which is $n= n_1 + n_2$

- Program Length which is $N\hat{} = N_1 + N_2$
- Estimated program length: $N = n_1 \log_2 n_1 + n_2 \log_2 n_2$
- Program Volume: $V = N * \log_2 n$
- Program difficulty: $D = n_1/2 * N_2/n_2$
- Programming effort: $E = D * V$
- Estimated time: T = E/S seconds. S is the speed of mental discrimination. The recommended value is 18.

6. Object Oriented concept based Metrics

   a. This is the most widely used metric that is used in software defect prediction [41]. Some common defect prediction metrics are as follows.

      i. Coupling Between Objects (CBO) [42]

      When a class consumes other class's methods or variables, there is a coupling between those classes. The number of classes a class is coupled with is considered.

      ii. The response of a class for a class (RFC) [42]

      When an object of a class invokes the methods of the class, those methods can call other remote methods. RFC of a class is the sum of the number of methods in the class and the number of external methods called by the methods in the object.

      iii. Message passing coupling (MPC) [42]

      Within a local method of a class total number of call statements identified in a class or the number of messages passed between objects.

      iv. Data abstraction of coupling DAC [42]

      This is the number of other class types defined in a class. This does not include the primitive types, inherited types from the base class, or the system types.

There was another effort on building a universal or common model of Defect Prediction [37] within a single project and across the projects. One motivation to go for a common model of Defect Prediction is to get rid of creating separate models for individual projects. This common Defect Prediction model removes the requirement to refit the models that are project-specific or release specific for an individual project. Understanding the primary relations between the given software matrices and the defects, a universal model can be derived. The problem with such a common or global model is the variation of the distribution of predictors. Thereafter, what they have done is they clustered the projects into 26 predictors comparing the similarity of the distributions and get the transformations of the rank with quantiles belong to a cluster. They further apply that universal model on 1398 number of open source projects that are available in GoogleCode and SourceForge. The universal model got equal prediction results to the individual external five projects that used project-specific models.

Cross-project defect prediction they have considered as an approach to start towards a global model of Defect Prediction. A difficulty when building cross-project defect prediction models can be linked to the distribution of predictors [38]. Their solution for that possible problem is they followed two solution approaches.

1. Using the projects that have similar distributions to the targeted project. This uses a partial data set and results in multiple models.

2. To make like the distribution of the target and training project, transform predictors. This approach binds to a pair of training and test data sets.

Further they got that software metrics vary with the project context. Therefore, combining all the insights, they attempted building a global model of Defect Prediction for a large set of projects that have diversified contexts.

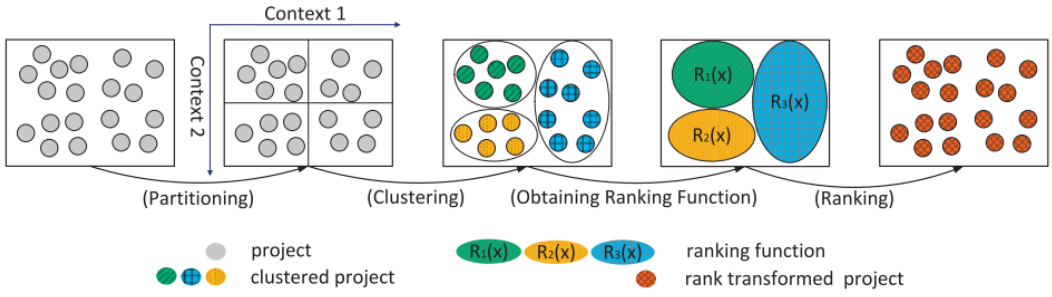

Figure 2.8: Four-Step approach for Universal Defect Prediction Model [38]

Different projects from different contexts show different distributions [39]. Therefore, as a solution to this problem context-aware transformation of rank approach as shown in figure 2.8 is proposed.

The four steps are explained as follows.

1. Partitioning is creating nonoverlap grouping based on the six different context factors. The six context factors used as predictors are as follows.
    a. The language used for Programming
    b. Tracking of issues
    c. The sum of code lines
    d. The sum of files
    e. The sum of commits
    f. The sum of developers
2. From the project groups, Cluster projects that have a similar distribution of predictor values. They have assumed that projects with similar context factors have similar software matrices. Also, the projects having different context factors have a different set of distribution of metrics. Used code and process metrics are as follows.

Table 2.3 List of Software matrices used.

Type	Metric Level	Metric Name	Description	File Level
Code Metrics	File	Loc	Line of Code	value
		C_L	Comment Line	value
		N_{STML}	Number of Statements	value
		N_{FUNC}	Number of Functions	value
		R_{CCd}	Ratio Comments to	value

			Codes	
		M_{NL}	Max Nesting Level	value
		Wmc	Weighted Methods per Class	avg, max, total
	Class	D_{IT}	Depth of Inheritance Tree	avg, max, total
		R_{FC}	Response For a Class	avg, max, total
		N_{OC}	Number of Immediate Subclasses	avg, max, total
		C_{BO}	Coupling Between Objects	avg, max, total
		L_{COM}	Lack of Cohesion in Methods	avg, max, total
		N_{IV}	Number of instance variables	avg, max, total
		N_{IM}	Number of instance methods	avg, max, total
		N_{OM}	Number of Methods	avg, max, total
		N_{PBM}	Number of Public Methods	avg, max, total
		N_{PM}	Number of Protected Methods	avg, max, total
		N_{PRM}	Number of	avg, max,

			Private Methods	total
	Methods	C_C	McCabe Cyclomatic Complexity	avg, max, total
		F_{ANIN}	Number of Input Data	avg, max, total
		F_{ANOUT}	Number of Output Data	avg, max, total
Process Metrics	File	N_{REV}	Number of revisions	value
		N_{FIX}	Number of revisions a file was involved in bug-fixing	value
		AddedLoc	Lines added	avg, max, total
		DeletedLoc	Lines deleted	avg, max, total
		ModifiedLoc	Lines modified	avg, max, total

3. For each cluster, form a ranking function with the 10^{th} quantiles of predictor values. This will address the large variations that the distribution of predictors has.

4. Convert the predictor raw values to one of the ten levels by applying the ranking functions.

Based on the work they have done towards a Universal Defect Prediction Model, they are evaluating the feasibility of the model to use in commercial projects. And extending this model as a plugin for the Integrated Development Environment (IDE) to alert developers on risks at the early stages of the Software Development.

2.2.4 Comparing Research Solution with the Literature Review

This literature review discusses the several Defect prediction methods and several approaches discussed to evaluate Model-Driven Engineering. Data Science-based solutions are much highlighted when considering the Defect Prediction Methodologies.

This research problem is understanding defect prone designs in the Software implementation phase that can throw errors in the Software maintenance phase. UML Class diagrams have chosen to be evaluated when they are designed in the Software implementation phase. The importance of Software Design Principles has considered here and its potential of preventing Software defects when UML Class diagrams are designed is majorly considered here as the defect prediction mechanism in the Class Diagrams.

Throughout this Literature Review, there is no research found as an analysis of the potential of predicting and preventing Software Defects using Sofware Design Principles.

This research will uncover the hidden potential of defect prediction in the Software Design Principles and analysis of the Software Design Principles in the UML Class diagram that is used for the Software implementation phase will be the methodology of defect prediction and prevention.

Chapter 03

Proposed Methodology

3.1 Solution Architecture

It is important to understand how the Defect Prediction can be well suited with Model-Driven Engineering to reduce the possible defects that can be occurred from the Model-Driven Engineering based approaches.

Model transformation in between the different models among different abstract levels is something straight forward. In this research there will not be an objective to embed a Defect Prediction Model in between different abstract levels of the Models used and transformed in Software Engineering.

Instead a validation is proposed to the data that are given as the inputs to the Model-Driven Engineering. This validation is facilitated via the proposed Defect Prediction Methodology. In this research, we propose to apply defect prediction methodology once the Model transformation is done and ready for a full implementation level and a complete solution.

In Object-Oriented Design, there are Design Principles that act as the pillars for the Design Patterns as well. These Design Principles are very important in Object-Oriented designs. We can see that the design disciplines based on Design Principles prevent predictable issues in the source code.

In the current competitive business world, the Requirement engineering part is a crucial role. From the stakeholder's point of view, requirement changes and feature additions come frequently. Possible reasons can be as follows.

- Business aspects and expectations can be varied in shorter periods. The current world is a competitive, much closer world. Therefore, business solutions may not be long-lasting solutions. Competitors need additional features and enhancements frequently to compete high in the market.

- A better solution comes after several revision cycles. For the stakeholders, they can describe the problems they have and hint solutions, however they do not have the idea of the exact solutions. When they go through several revision cycles, they gradually understand what they need exactly. Therefore, Agile software development methodology is a better approach rather than the Water-Fall method. The prototype-based approach in shorter sprints is a more

practical approach rather than an approach with finalized set of requirements which is an impractical target.

When you come to the solution design aspect, then it becomes the fact that Changes in the software are inevitable. If the designer cannot understand the best practices of absorbing the changes into the source code and making it stable, the solution will become fragile and adding the risk of failing the entire business.

Now it is clear that we must entertain changes and supporting methodologies are crucial in Software Engineering. Let's examine into which extent the Design Principles support the absorbing changes into the source code design. To approach the solution let's take the following steps.

1. Understand Design Principles.
2. Understand the importance of Design Principles when absorbing the changes into the source code.
3. Understand how future defects are predicted when the UML Class diagram is violating the design principles.

Let's get back to those steps individually.

3.1.1 Design Principles

The following are some important design principles that should follow in the Design of a Software System.

1. Separate or encapsulate the aspects that can vary in your application from the once that stay the same [17].
    a. This design principle is valuable in keeping a better extendibility in the source code. The business Aspects that can be varied are the direct impactors in a situation where extendibility is expected. Business aspects are explained as the business sub-components that can be changed in the future and it can be predicted by the time system is designed. For example Vehicle design is a business aspect that has the possibility of change or upgrades in the future. Separate them and encapsulating them allows a proper mechanism to handle the variations of those aspects.

2. Program to an interface, not to an implementation

    a. This principle explains that when there is an entity and there is another who consume it, both of those should rely on abstractions.

3. Favor composition over inheritance

    a. This principle also enhances the possibility of extending the code. Where the inheritance is there, parents and the child classes are binds in a way that all the changes apply to the parent class will be reflected in the child class. However when using the composition, the dependency between two classes has been reduced and changes in either of the class have less impact on the other class.

4. Let the objects that interact with each other loosely coupled.

    a. This principle also allows to the changes and the extendibility of the code. When they are more loosely coupled, it is easier to maintain the code. The amount that the coupling is tight, changes into those combined entities tends to break the existing functionalities.

5. Classes should be open for extension, closed for modification.

    a. This principle also explains the importance of keeping the components of the source code well separated according to the responsibilities they have assigned. In that case, classes are needing to be changed, we can extend those into another class that will not allow us to break this rule.

6. Depend on abstractions. Do not depend on concrete classes.

    a. Again, this principle explains the importance of loosely coupled dependencies between the entities. They should rely on the abstractions in between them.

7. Principle of least knowledge. Talk to your immediate friends.

    a. It is important to let the classes invoke the methods from the following ways.

        i. From the object itself

ii. From the objects passed as the parameters to the object.

iii. From any object that the class instantiates.

iv. From an instance variable of the class.

Calling distance classes that can be accessed via the reference variables that tend to make tight couplings are disallowed from this design principle.

8. The principle of Hollywood. Do not call us, we will call you.

    a. High-level components should be the entities that define when and how the low-level components are accessed. However low-level components should not call the high-level components.

9. Single Responsibility – A class should only have one reason to fail.

    a. A class should be responsible for one thing and for that thing only. If the class is changed, that will be for that one single thing that is maintained in a class. For any other reason that class should not be changed. Because of the one single purpose of the class, it should be failed. Not due to another reason/s.

3.1.2 Importance of Design Principles when absorbing changes into the Design

In this section it is targeted to understand the defect preventive potentials in the maintenance mode of a Software solution when the Design Principles are followed. When software requirements are changed or new requirements added, this will show how each Design Principle will help to absorb the changes.

Understanding the business aspects that can vary and encapsulate them.

In the early design phases, business entities can be identified. Such business entities can be extended towards the source code as well. Even in early design phases there is a varied possibility of identifying the business aspects that

tend to change or add more in the future even though such additional is not appearing yet. Those can be predicted easily.

For an example of an educational material manufacturing business domain, the type of educational material is a business aspect that can vary with time. In the early days even before the digital era there is paper-based material. With the emergence of the internet and the higher usage of computers, the digital educational content media types have emerged. Then the paper-based content converted into digital media formats like HTML, PDF, etc. Other transactional or supportive contents like Word and Presentation type of formats also need to be supported. Therefore clearly the content type is an aspect that can change or vary in the educational content-generating business domain. It can be predicted at the early stages of the solution design.

We can get the advantage of Design Principles to cater to this changeable business aspects in a way that there is no restriction to change.

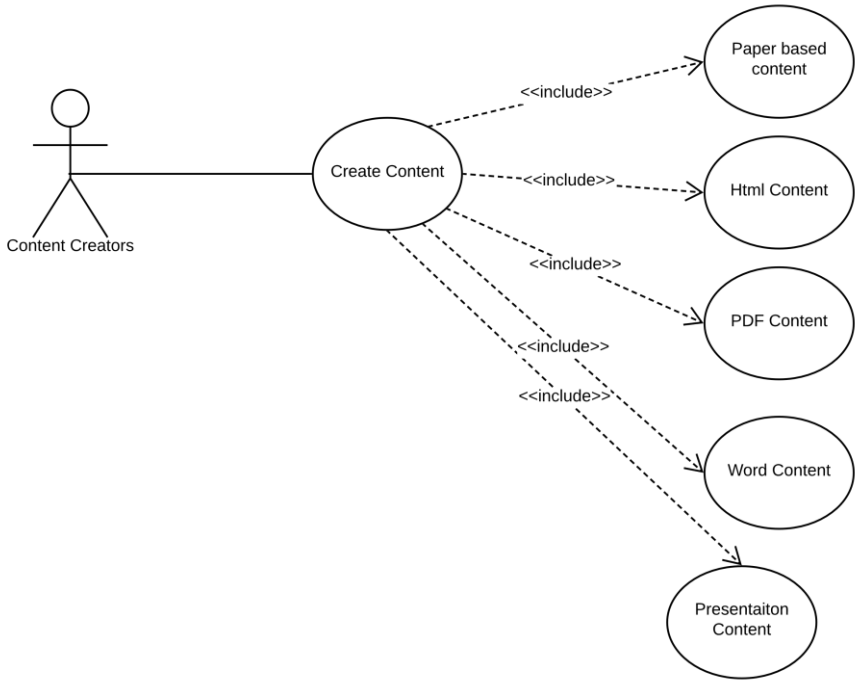

Figure 3.1 Content types created by Content Creators

In a continuously evolving technology world, educational content also improving. The current trend is adding Augmented Reality and Virtual Reality into the world of education. In that case we need to make the educational content ready for those formats as well.

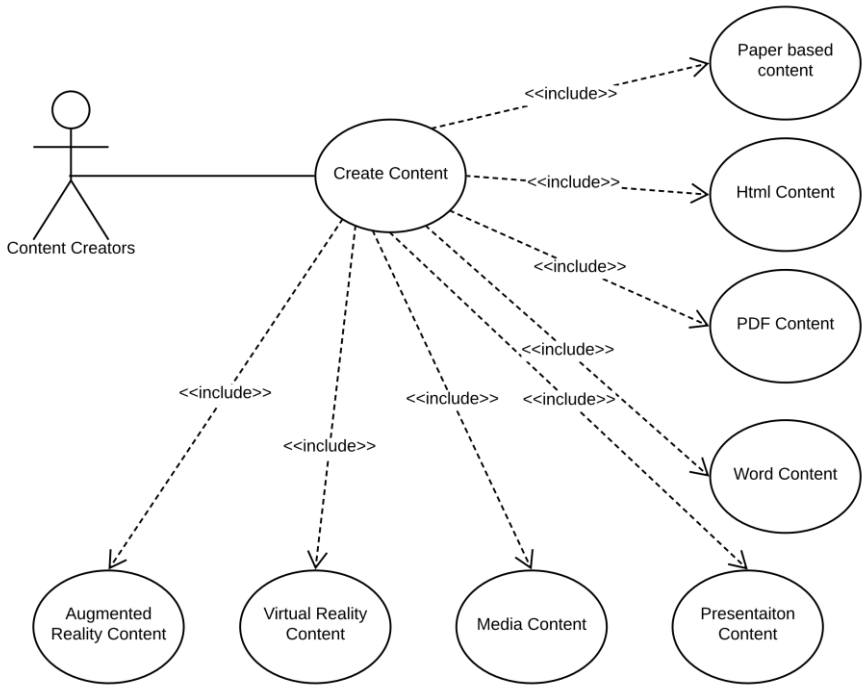

Figure 3.2 Content types of future variations created by Content Creators

It doesn't matter how many content types are introduced as the content type, as far as the educational content has identified as an aspect that can vary along with the business continuation, source code is ready to absorb any type of content creation.

When the intended behaviors are exposed through a certain interface however does not matter how the implementation is done as far as it satisfies the interface definition, it is called the encapsulation of a business aspect.

Program to an interface, not to an implementation

The next Design Principle is Program to an interface, not to an implementation. We need to understand the high-level abstract idea of this principle. After identifying the business entities, there will be interactions

between those business components. However most importantly these interactions happen in the source code that will be changed during the entire business stay in the market.

For example, a use case in the education domain that create content and export them to different formats for different business usages.

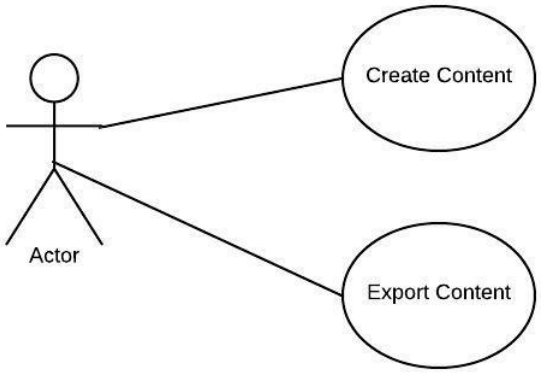

Figure 3.3 Main features done by the content creators

The changes in the business entities should happen in a way that the business entities interacting with each other are not impacted or lightly impacted. One way to achieve this is to introduce a contract/agreement or a discipline between the business entities interacting with each other.

One recognized way of doing this is keeping an abstract layer between the two business entities. The abstract layer can be an Interface in Object Oriented Programming languages like Java or C#. Those interfaces can define one or multiple method definitions. However most importantly not the implementations. This is how the interface defines the discipline. Any other class that implements that Interface, must build its implementation of that Interface definition. Business entities can interact with each other and change themselves as far as that discipline is not violated. The program itself validate for that though an efficient usage of Interface implementation for Object-Oriented classes.

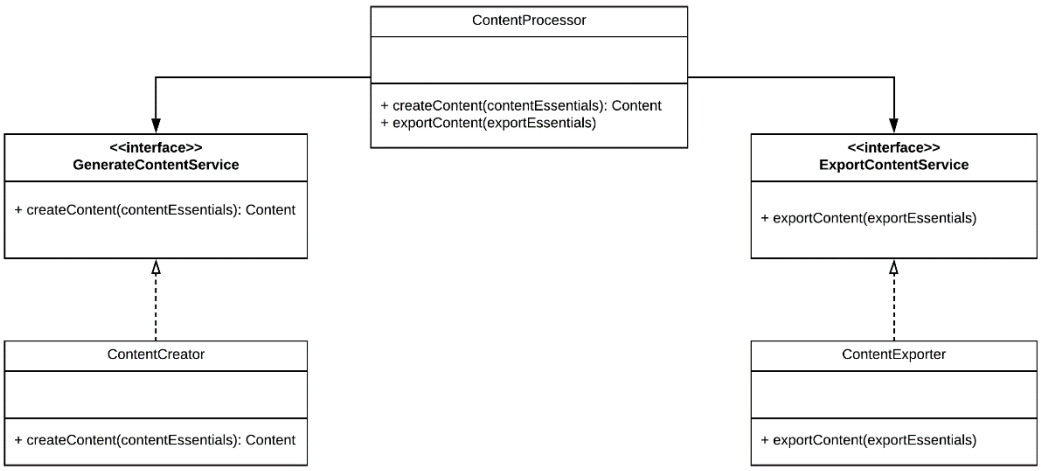

Figure 3.4 A class diagram with needed abstraction layers in place

Favor Composition over Inheritance

Inheritance is very popular among developers. Maybe it is one of the first concepts they learn in Object-Oriented Programming. And it provides the facility of reusing the features of the parent class among whatever number of child classes inherit the parent class.

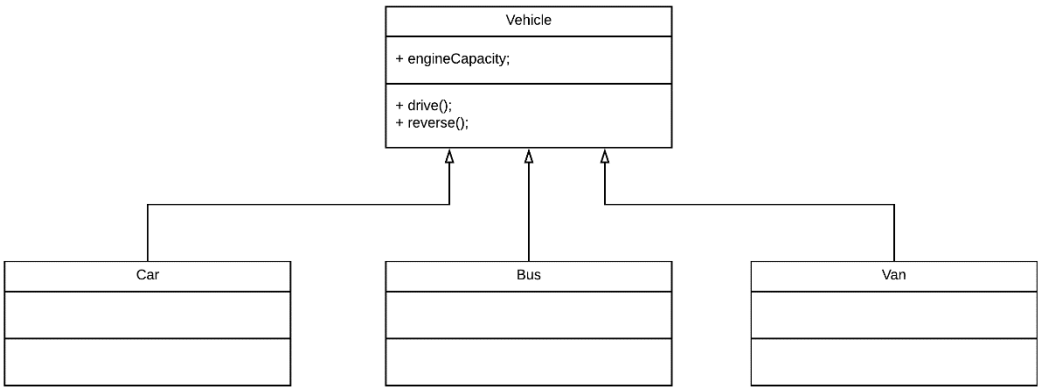

Figure 3.5 A Class diagram with inheritance demonstrated

Compared to inheritance, composition gives much more flexibility in implementing the changes on top of the existing code base or in the maintenance mode.

UML Composition has two types. Composition and Aggregation. When two entities are identified under Composition, and when one entity cannot exist without the other one, it is Composition and if the other entity can exist without the other entity it is Aggregation.

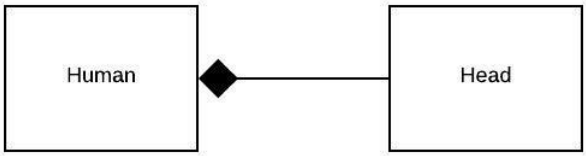

Figure 3.6 UML Composition

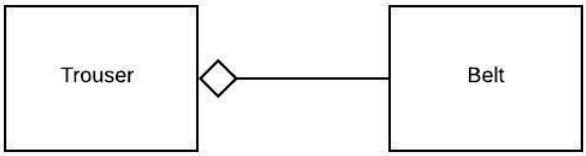

Figure 3.7 UML Aggregation

The native set up of the composition allows us to add any number of compositions around a business entity. Therefore, the composition is a suitable option for maintenance mode. A possible disadvantage of Composition is there is a possibility of reducing the code reusability. To prevent its code must be modular as much as possible and get rid of that behavior.

When we take the maintenance aspect, in Composition there is freedom for each entity to grow, since they are coupled to get some functionality done. It is a subset of an entity. In inheritance, the whole properties and the behaviors of the parent class have appeared in all the subclasses. Even though child class can change in the maintenance mode without impacting its parent classes, parent class's even a single change will appear in all the subclasses. It is counted as an advantage of inheritance, however when it comes to specific changes, it will add a lot of fragility into the source code.

Let the objects that interact with each other be loosely coupled.

The coupling can be best described when we consider the aspect of change. Coupling is two types as loosely coupled and the tightly coupled. When a change applying to an entity that is coupled with another entity, the second entity is impacted when the two entities are tightly coupled. It gives a lesser room for change within the expectation of those entities remain the same. Coupling applies not only in the UML classes, however, the layers of the solution and between components as well.

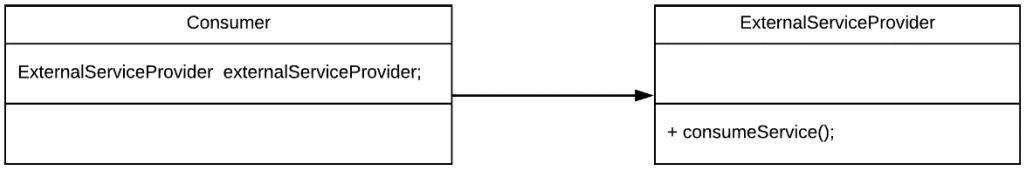

Figure 3.8 A demonstration of tightly coupled two classes

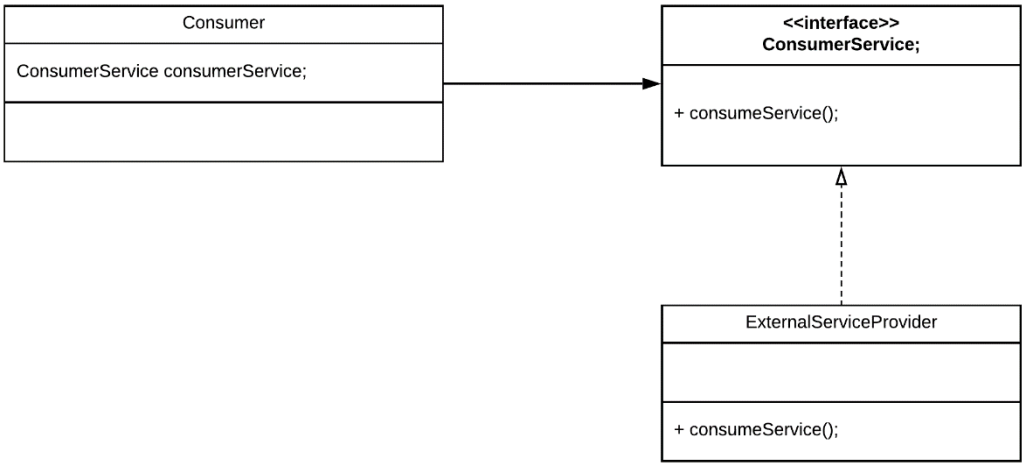

Figure 3.9 A demonstration of loosely coupled two classes

Classes should be open for extension. Closed for modification.

This is known as the Open-Closed principle as well. This explains that the classes should extend on the responsibilities assigned to that class should not modify for external modifications other than the responsibilities assigned to

that Class. Classes represent a single business entity or unit. Once a business entity is identified, it should be extended with changes that match its responsibility rather than they are modified for different responsibilities. Because modifying the existing business entities will impact the existing behaviors or responsibilities. The entity can be defined closed when it is maturely defined and stable or its responsibility is properly assigned. The extension can be applied in a way that composition is used along with the business entity.

For example let's have the business entity or Class Course and Class LectureRoom. The course is conducted at the LectureRoom. Assume there is a new requirement that comes to conduct Course lectures online. While we cater to this requirement inside the existing LectureRoom Class, it has to go beyond its assigned responsibilities. Assigned responsibility is conducting a Course in a physical Lecture room. In this new requirement, the Online Lecture room has its unique behaviors. When applying Open Close Principle for this example, instead of modifying the LectureRoom class, a new Class OnlineClass should be initiated to extend this new requirement.

Depend on abstractions. Do not depend on concrete classes.

This rule also encourages coupling loosely in between business entities. Accessing a concrete class should be done through an abstract layer to decouple the business entity. That abstract layer will define the contract or the discipline that should communicate between two entities. Therefore, depending on abstractions always lets to minimum impact when changes are applying or in the maintenance mode.

The design principle Program to an interface, not to implementation also interprets the same meaning of this design principle and encourages to consume other business entities through the abstraction layers.

Principle of least knowledge.

This is known as the Law of Demeter as well. This is a more scoped version of loosely coupling. A module should know the details or the references for closer modules. Not far modules. You simply need to have closer friend details. Not about the stranger's details. Only deal with the immediate friendly modules.

The impact of this rule is if a module or a business entity is changed, the impact will be predictable and lesser. It will be confident that far away modules are not going to impact when this principle is restored as a discipline. In Maintenance mode this provides a larger value to maintain the business modules non-fragile.

Principle of Hollywood.

This principle simply says Do not call us. We will call you. This is helpful when working with a set of external constraints like an external framework. According to the framework definitions, classes or interfaces has to be placed and the framework will call the necessary components.

Single Responsibility

This is a very important principle when isolating business entities and even granular components. Understanding the business aspects and assigning responsibilities will become crucial in implementing this. On the other hand, executing other Design Principles will become easier when the Single Responsibility rule is properly implemented.

Simply an entity, module, or component should hold its assigned responsibility and none other else. If responsibility is upgraded, using the Open-Closed principle you can define another module and connect to the primary module. It will allow us to extend the responsibility without modifying the existing module. When a change to an entity has arrived, it can be tracked as a different aspect or a different responsibility and can cater to encapsulating that responsibility and implement it.

3.2 How the Design Principles can be used as the Defect of Preventive methodologies.

A defect can be expressed as a functional or nonfunctional expectations breach in the software module. The value of the usage of Design Principles comes when the source code is completed, and unit tests are there for the code verifications. Then we can monitor the functional expectations against the changes apply to it. Design Principle's values are becoming to see the value of it when seeing how much they support the changes into the software.

3.2.1 Importance of Object Constraint Language

Object Constraint Language has originated to overcome the limitations of the UML. It helps to understand the details aspects of the system design.

3.2.2 Possible Solution approaches

Data mining and Machine Learning-based approach

In the field of Defect Prediction, Data mining and Machine Learning-based approaches have a greater capacity. Chapter 2 also explains the previous Data Mining and Machine Learning-based approaches followed and how suitable that concepts into the Defect Prediction.

This research will allow to keep the existing MDE approaches as it is. Identifying the different abstraction levels to fill the problem solution gap and MDE will be used to fill that gap. When the closest model to generate the code or implement the code is generated, we can apply the proposed Defect Prediction methodology.

The proposed methodology can be one of the following.

1. Propose a new Defect Prediction methodology that suites the MDE.
2. Combine an existing Defect Prediction methodology that will be much suited to the MDE and propose it.

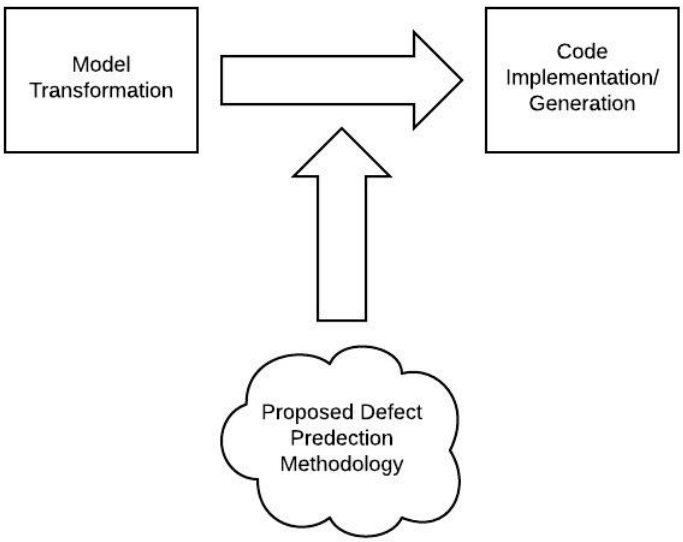

Figure 3.10: High-level solution plan

Evaluating the MDE Model against the Design Principles

Now we have aware of the impact and the value of Design Principles in preventing issues in the source code in the long run of the solution or the maintenance mode. If it is possible to validate for Principles violations by the time of Models or the business entities created, we could have prevented defects that could occur in the maintenance mode requirements adding and changes. An agent who track the model designs and check for any Design Principle violations and alert.

This research has identified two types of solutions to this.

Shared back end solution

A shared back end solution can be used with any type of consumer. How the data are to be communicated between the consumer and the service is documented in a JSON schema.

Front end solution

Users are involved in creating business models in a graphical tool. If the users can be informed of any Design Principle violation in their design steps it will be a more proactive way of informing them. We can use the increasing

usage of front-end technologies for this solution and give a better understanding of the Design Principles.

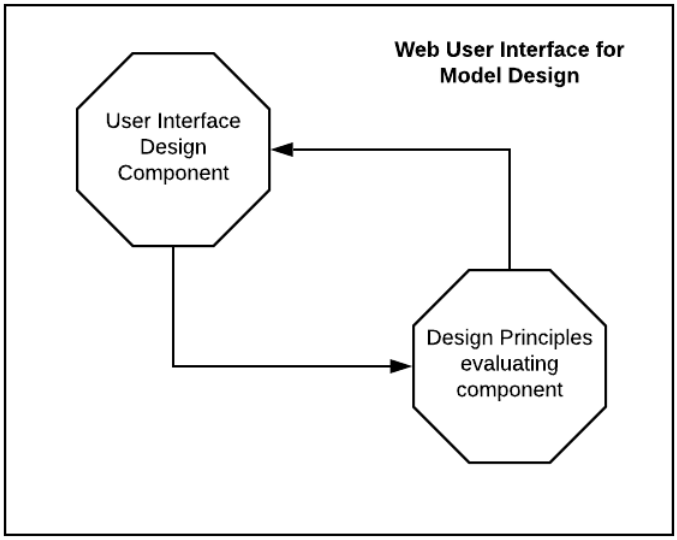

Figure 3.11 How modeling UI is linked with the Design Principles evaluating module

The usage of front-end technologies will be helpful in this. There are rich front-end web libraries are emerging now. A single page application solution can cater to a solution like this.

3.3 Selected Solution to be implemented as the solution for Research Problem

Evaluating the Model-Driven Engineering model against the Software Design Principles selected as the solution for this Research problem. Shared backend solution has selected as the solution over the front end web solution considering the following advantages.

1. A shared backend solution can be consumed by several consumers. It will provide the flexibility of usage between several consumers.
2. Processing will happen in the backend, it will decouple the completion of the non-functional requirement from the consumer, rather the solution is processed in the front end and the consumer has to take care of the processing.

Chapter 04

Solution Architecture and Implementation

4.1 Solution Introduction

As mentioned in the previous chapter, the solution will apply in the step of building or generating the code from the immediate model. There will be an external mechanism that will be introduced to do the predicting of the defects in the model. Therefore, the model should be sent to that external module in an accepted format for it. Then that external module assesses the model and it will return the possible issues of the model.

4.2 Characteristics of the external module

1. This module will be an externally hosted microservice.
2. There will be a JSON schema that defines the message format that is accepted by the microservice.
3. The model needs to be converted into a JSON message following the JSON schema.
4. Converting the class diagram into the JSON message is currently not under the implementation since it is out of scope.
5. The current implementation includes the assessment of the Class diagram of a system.
6. The current implementation will evaluate the JSON message for class diagrams against the design principles.

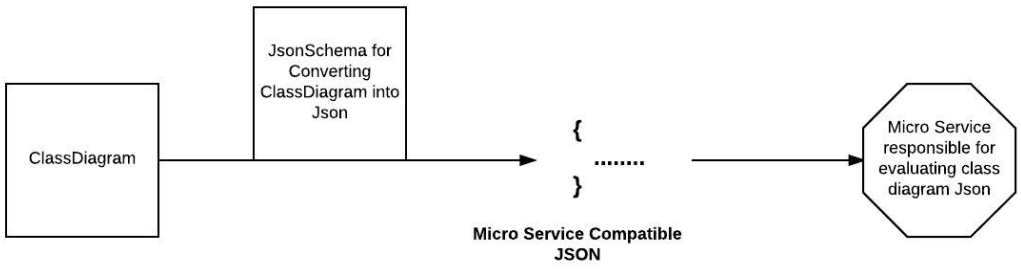

Figure 4.1 High-level Design of the solution

4.3 Concepts behind the external module

There are known design principles and patterns that always drive to better designs and implementation which always encourage extendibility, readability, and maintainability of the code. Violating such design principles leaves a clear space to trigger any defect in any moment.

Even the value of design principles and design patterns is clear, most of the time those design principles and patterns tend not to be used. The pressure of the business expectations that expecting the fast delivery more focusing on functional requirements however not much the nun functional requirements and the best practices that could save a lot of future time that will be consumed.

4.4 Detailed Design of the solution

This solution contains the following components. Together with those components, there will be a service that responds to various design requests, without any dependency on the platform that the design is implemented.

- Microservice
    - This service will absorb the design evaluation requests and process those requests to make the assumptions and predictions of possible defects in the system that can occur in the short term and long term. This service is a REST-based service. There are REST endpoints along with the explained steps to access those. /api/v1/classmodel is the current REST endpoint that serves the design requests.
- JSON Schema
    - This schema defines how communication between the graphical design and the microservice should happen. This schema has explained the JSON properties that are sent to the microservice, and how they should send. For each property there is small documentation that the users can understand and adhere to.
- JSON Payload

- This JSON payload is created according to the JSON schema and will send along with the request body of the REST call that is made to the microservice.

- Response
  - The response also a JSON output. The requester can use this response and properly show it in its design diagram.

4.5 The current implementation of microservice

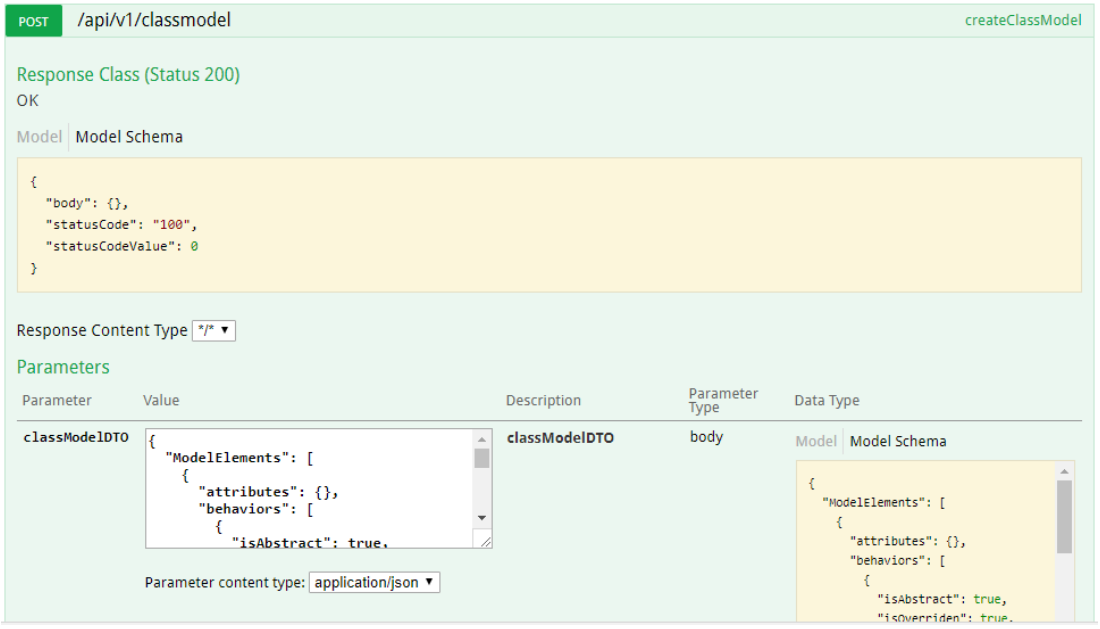

Figure 4.2: The swagger interface of the solution service

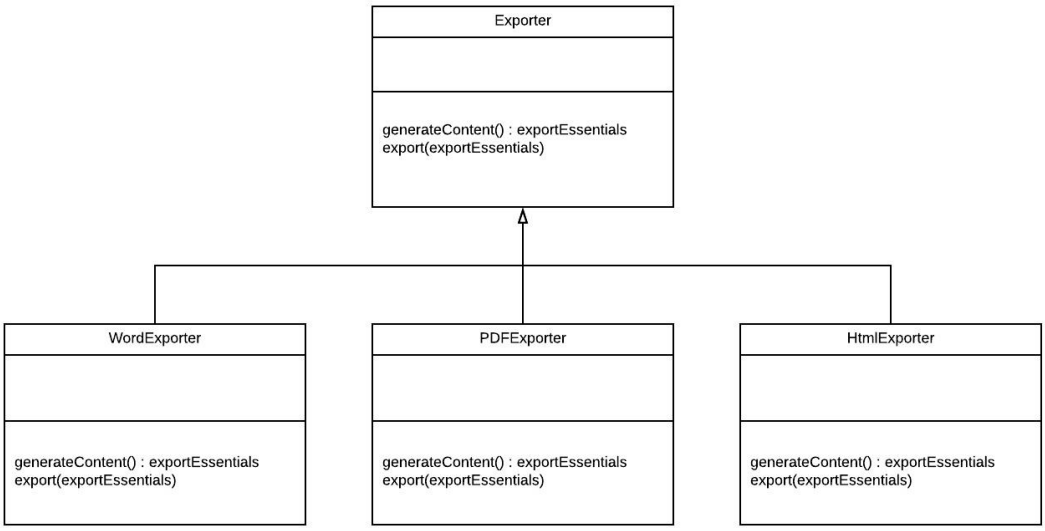

Figure 4.3: Sample design diagram that is submitted to the solution service

A sample class diagram that is submitted for this system for defect analysis.

A sample response from the service when a design request is made.


```
[
  {
    "_message": "Following classes are not associate with abstract layers. WordExporter PDFExporter HtmlExporter"
  },
  {
    "_message": " Try to use Composition over inheritance in order to make the design more flexible."
  },
  {
    "_message": "generateContent and export are aspects that vary. So they can be seperated."
  }
]
```


Figure 4.4: Response from the solution service for the submitted design

When explaining the core algorithms used in the solution service to evaluate the class diagrams, the Following steps happen.

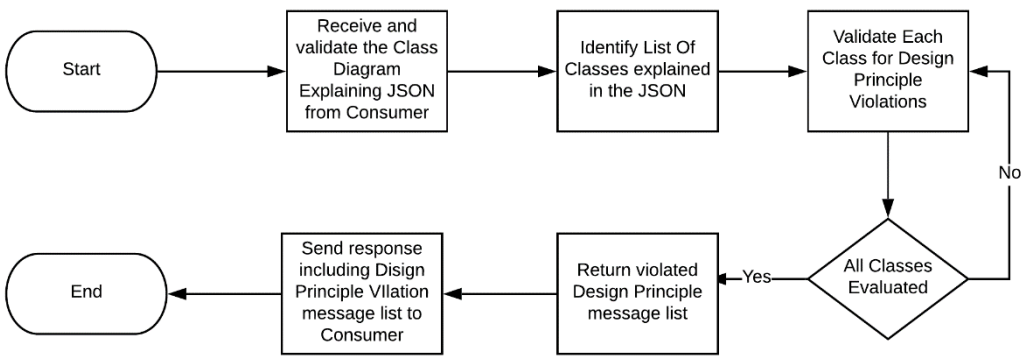

Figure 4.5: Flow of the Solution Serice

1. First this solution module will take the JSON data from the consumer and validate before sending them for evaluation.
2. Then it will identify the individual class explanations in the JSON.
3. Then it will identify Design Principle violations in each Class JSON.
4. After evaluating all the classes, it will send the design violation messages back to the consumer.

4.6 Deployment of the microservice

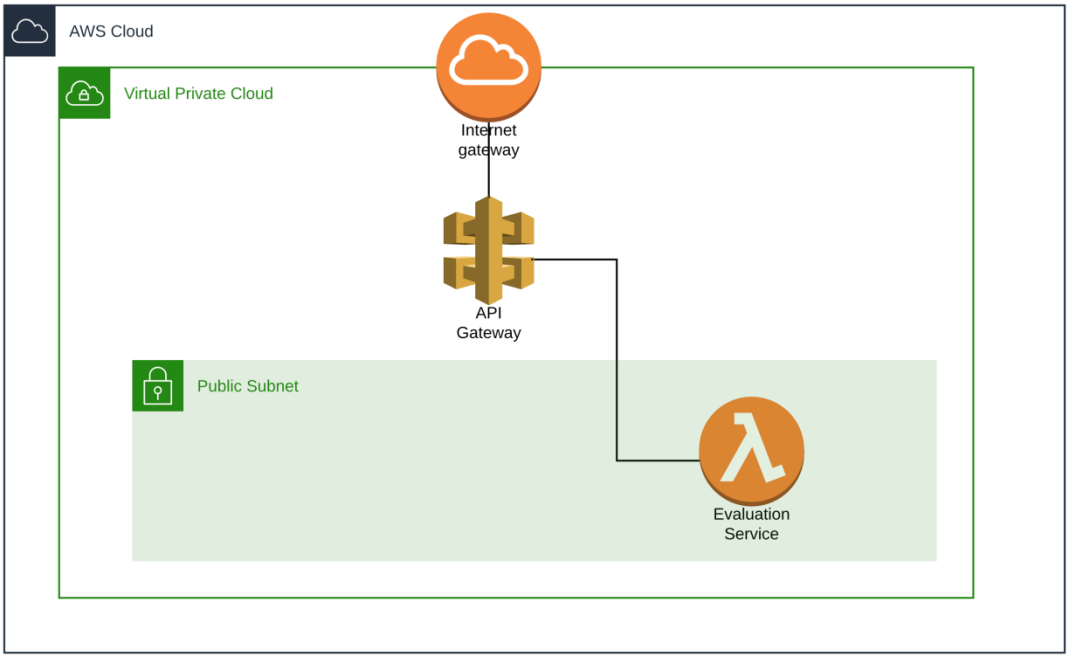

Figure 4.6: AWS Cloud deployment of the Solution Service

This solution is hosted in the AWS cloud environment. This service is hosted as an AWS lambda function. In nature AWS lambda functions are the candidates of Function as a Service cloud service and it is hosted in the cloud environment. No dedicated hardware is allocated to the lambda function, however once there is a consumption of the lambda function, lambda function creates its execution containers in the cloud and scale it according to the rate of the requests are getting. Based on the requests rate, the lambda function execution containers can be increased or decreased. If there is zero consumption, there can be no execution containers.

Therefore, there is no worry about the availability of this solution which is under an AWS lambda function.

4.7 Further possible extensions of the solution

1. Implement the generating of JSON string against the JSON schema exposed from a class diagram. Currently this part is done manually, and it will be an additional effort for the Software Development that consumes a separate effort.

2. Integrate this microservice with web-based diagram drawing tools. For example, MxGraph [18] JavaScript library can be integrated with a tool like this and add some intelligence to the class diagram.

3. This solution will reside as an open service. Therefore, it is needed to protect this service from security threats. A suitable authentication mechanism should be merged into the system to provide a trusted service.

4. Extend the module and integrate the machine learning technique to learn from the data that is sent to the module and do more predictions with significant accuracy.

Chapter 05

System Evaluation

5.1 Introduction

This solution is important in Software Development as a disciplinary guideline. Most of the time the Software developers with the delivery mindset cannot invest in the best design concerns. Teamwork is the building strategy for most of the Software developments. In that case few developers work on the same module most of the time. In those cases, there is a rapid development speed and the possibility to occur design violations is high. Design experts and Software Architects can drive-through towards the completion of a given design with manual interactions.

In some well-known Software Development approaches, there is a high possibility to introduce defects into the system and pay a lower priority on some good Design Principles. Software Development industry always has the competition with the time, budget, and the resources that can align with the business requirements. In Agile terms also, tasks are defined in business units and, only the given business unit must be achieved within the committed sprints. Therefore, needed Design principle embedding and the decisions of needed refactoring steps to keep the source code less fragile are tending to miss. It is a real disappointing side of this industry.

When we take the Junior individuals, who involve in software development in most of the cases, do not aware of the Design Principles and the importance of this. It also makes the situation worst. These patches damage the long run of a business.

This solution is a kind of automated design evaluating mechanism. Once a change is applied to the design, automatic design validation happens and alert the developers in the earliest stage of the development. It is quite important when development happens in a rapid phase. Earlier the defects prone places can be identified, waste that can add rather when they identify later, can be reduced.

5.2 Evaluating the system concepts

When we take two classes that are communicating with each other, it is always needed to reduce the coupling between them. An abstract layer is needed to be kept in between those two classes which define the contract or the agreement between those two classes. Other classes can also use that contract if they need to use it. When applying this concept, both classes can keep their behavior with less impact to

the other class. When the changes happen to the classes this is important to keep two classless fragile against the changes that apply to the system.

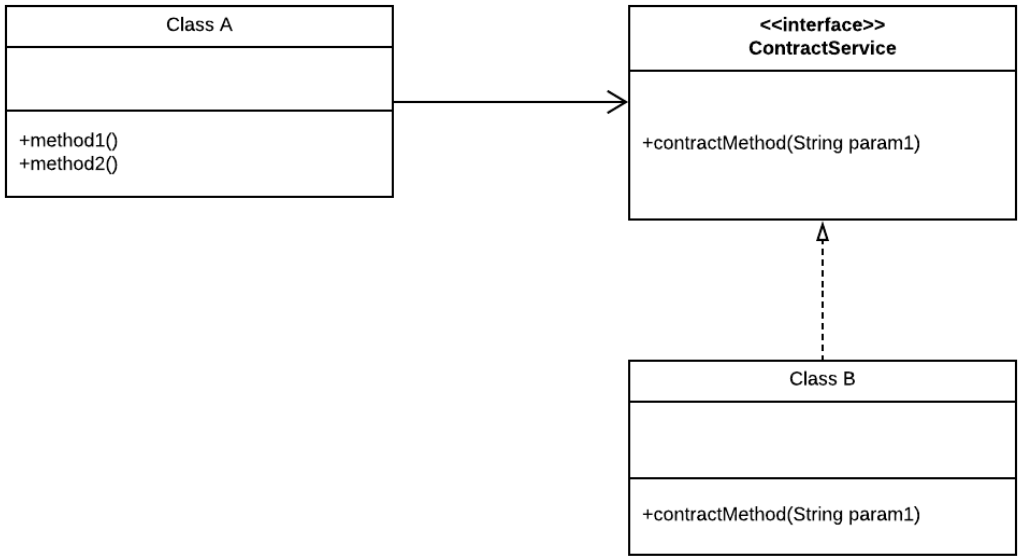

Figure5.1: An example of how the abstract layer reduces the coupling

In the above scenario ClassA access the ClassB using the ContractService. ClassA and ClassB can agree on the ContractService for communicating purposes. ClassA requests contractMethod from ClassB. In this case ClassA can change without any dependency on ClassB. ClassB can change without impacting the contract that is agreed via ContractService.

There are Design Principles to encourage this discipline in the codebase.

- Program to interface. Not to implementation.
- Let the objects that interact with each other loosely coupled.
- Depend on abstractions. Do not depend on concrete classes.

Dealing with changes in a system is the most important thing in design aspects. In the long run of the business, there are changes in the Software that needs to be added to. When adding changes to the existing software, it is needed to keep existing functionalities untouched as much as possible. There are certain design principles for that. Single Responsibility rule, Favor composition over inheritance are few of them.

5.3 Evaluating case scenarios

5.3.1 Evaluate an inheritance-based solution design

Let's take the following sample design and the design evaluation via the newly implemented service.

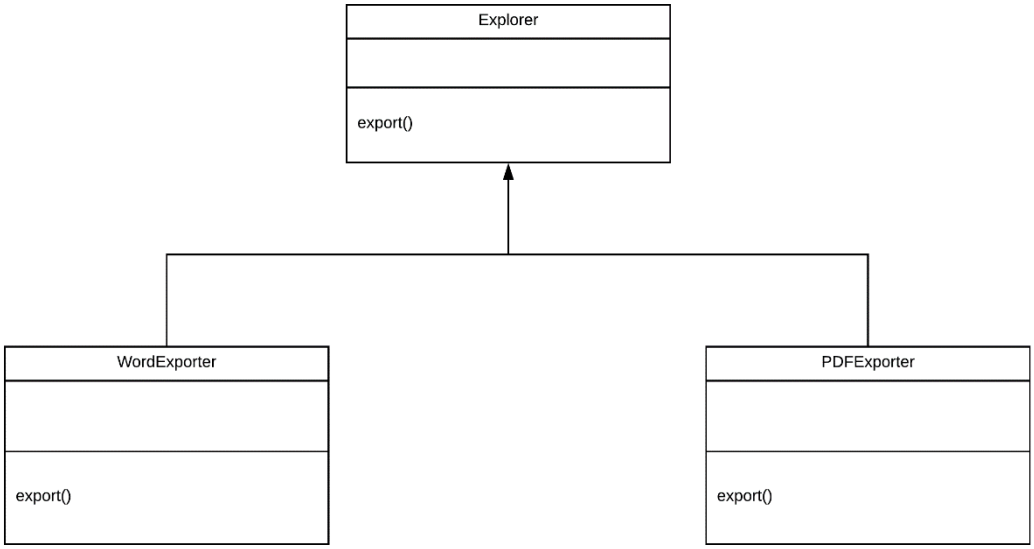

Figure 5.2: Sample design diagram that is submitted to the solution service

The above design has to be first converted into the service compatible JSON message with the help of JSON schema. Both are added in the Appendix section. Following is the result that is coming from it.


```
[
    {
        "_message": "Class Exporter has a inheritance dependancy with class WordExporter. Behavior/Method implentation may vary in the class child WordExporter"
    },
    {
        "_message": "Class Exporter has a inheritance dependancy with class PDFExporter. Behavior/Method implentation may vary in the class child PDFExporter"
    },
    {
        "_message": "Class Exporter has a behavior / Method export that can have deviated behaviors in the child classes. Better to refer it through an abstraction layer"
    },
    {
        "_message": "Class WordExporter has no dependancy with an abstract layer. Possible violation of Design Principle"
    },
    {
        "_message": "Class PDFExporter has no dependancy with an abstract layer. Possible violation of Design Principle"
    }
]
```


Figure5.3: Service output for the design

Exporter class is the superclass of WordExporter and PDFExporter. Therefore the inheritance itself provides a tight coupling between the parent class and the child classes. If we can keep an abstract layer, we can reduce the coupling between those.

When dealing with composition rather than the inheritance we always have the facility to introduce abstract layers and reduce the coupling. Changes also can be added to the system as composite components. In this way favoring composition over inheritance is a good design principle, which is indicated by this response.

In this design we can identify that export() method or behavior can differ slightly in the child classes. Those varying aspects need to be treated separately. Because those are identified as varying aspects and future also these can further change. Therefore, a non-fragile code needs to be ready for those. That status is indicated in the design response.

Based on the design suggestions we can change the design as follows.

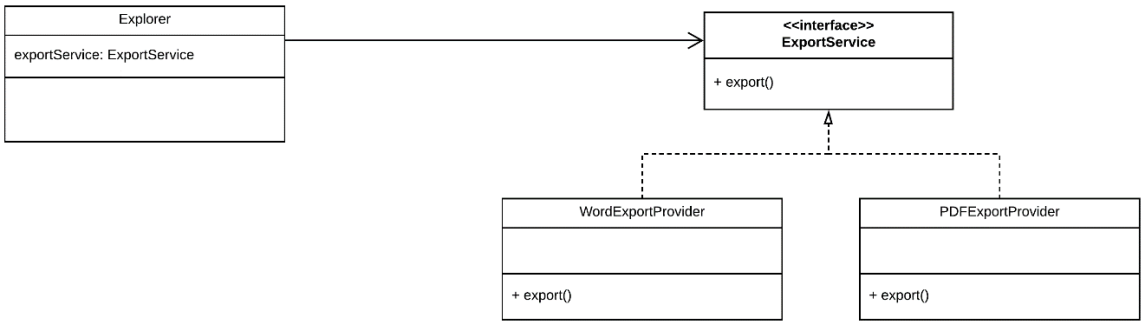

Figure5.4: Suggested solution after recommendations

In this design any number of ExportServices can exist. From the changing perspective any number and any type of export types are allowed. Any they can grow without impacting Exporter or its subclasses. Those subclasses can choose the needed export service for their functionality in the runtime most importantly. For content creation also, the same strategy is applied.

5.3.2 Evaluate Association based solution design

We can consider a diagram that uses the Association to link two business entities. A sample business case can be as follows.

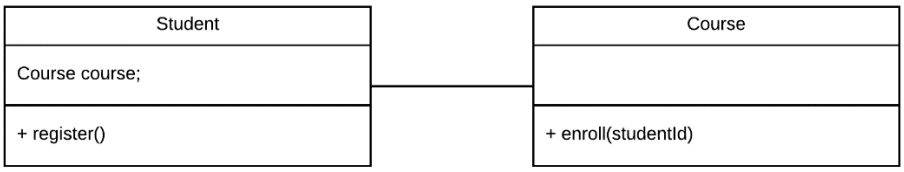

Figure5.5: An Association based solution design

In the above business, case student enrolls in a course. Two business entities Student and the Course, are linked together with the business function course enrollment. When we evaluate this design with the defect predictive solution we can see the following results.


```
[
  {
    "_message": "Following classes are not associate with abstract layers. Student Course"
  },
  {
    "_message": "Program to abstract layer for the following classes. Student"
  },
  {
    "_message": "Following classes depend on concrete classes. Student"
  }
]
```


Figure5.6: Service output for the design

After those results, we can suggest a design like this. There is an abstraction layer represented by the interface that is introduced in between the Student and Course Classes and the Student Class will associate with the abstraction layer representation Interface EnrollmentService.

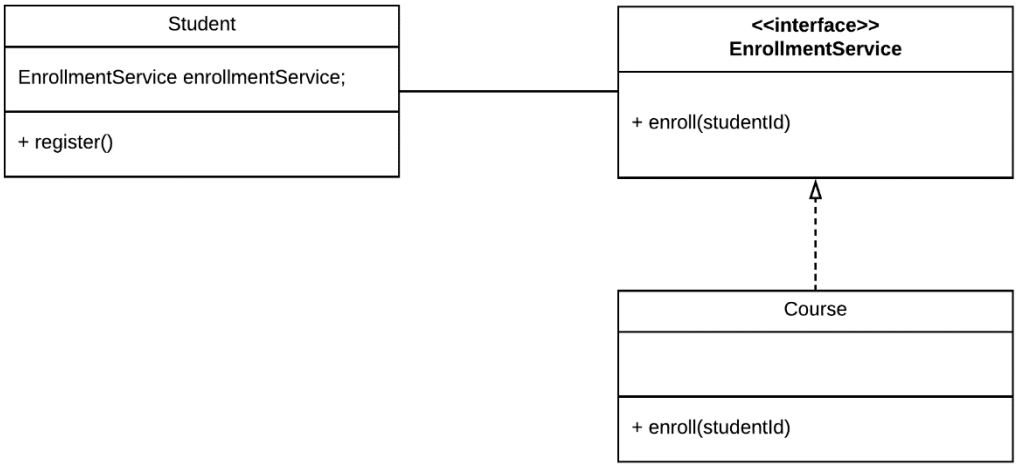

Figure5.7: Suggested solution after recommendations

5.3.3 Evaluate Composition based solution design

The Composition can be described as the HAS-A relationship. In the following relation, the course has course materials. Here the composition explains a compulsory relation between two entities that is a course that must have course materials.

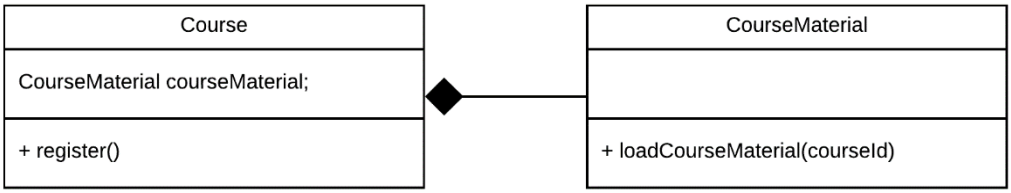

Figure5.8: Composition based design


```
[
  {
    "_message": "Following classes are not associate with abstract layers. Course CourseMaterial"
  },
  {
    "_message": "Program to abstract layer for the following classes. Course"
  },
  {
    "_message": "Following classes depend on concrete classes. Course"
  }
]
```

Figure5.9: Service output for the design

To decouple the integration between two classes, the abstract layer can be introduced as follows as per the instructions from the solution service.

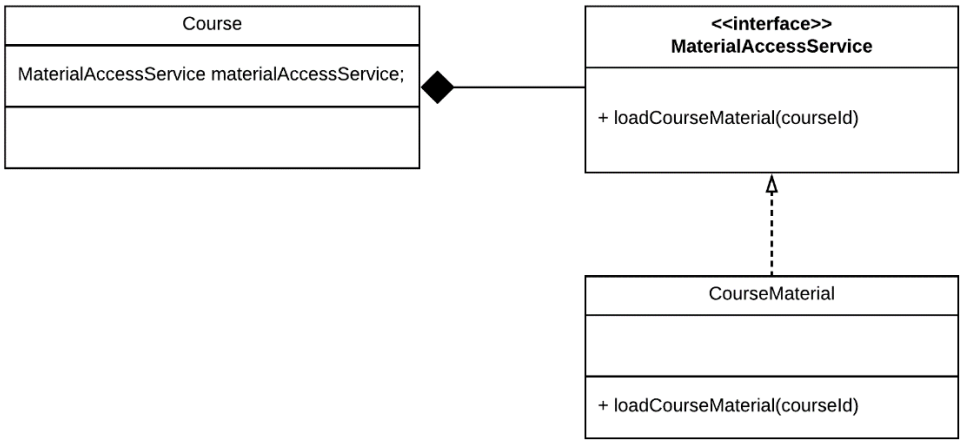

Figure5.10: Suggested solution after recommendations

5.3.4 Evaluate Aggregation based solution design

Aggregation is also notation of two entities that link through the HAS-A relationship. However having a sub-entity is not compulsory in this relationship. As per the following diagram, having a swimming pool in a school is good, however, it is not compulsory.

Figure5.11: Example of an Aggregation based UML


```
[
  {
    "_message": "Following classes are not associate with abstract layers. College SwimmingPool"
  },
  {
    "_message": "Program to abstract layer for the following classes. College"
  },
  {
    "_message": "Following classes depend on concrete classes. College"
  }
]
```

Figure5.12: Service output of the design

When the instructions are followed from the solution service, we can derive a class diagram as follows.

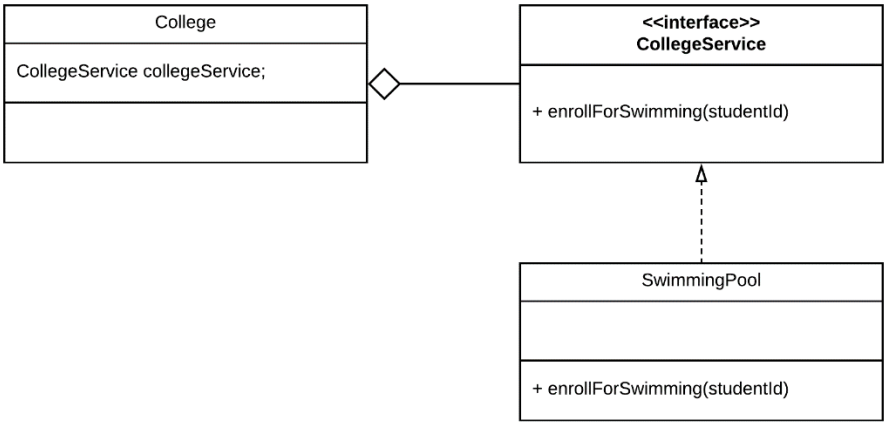

Figure5.13: Extension of the Aggregation based UML after evaluation

5.4 Evaluating system behavior

This solution will be hosted in an environment where any consumer can request for its validation service. Apart from the functional requirements we identified to be achieved from this module to evaluate UML class diagrams against the Object-Oriented design principles, there are a set of nonfunctional requirements that we need to identify to serve a better service for this module's consumers.

Those nonfunctional requirements are identified as follows.

1. High availability of the service
2. Performance measured by the Response time
3. Parallel requests it can handle
4. Scalability
5. Portability
6. Security

The system kept under more experiment cycles to test the above parameters.

This solution is hosted in the AWS environment as a lambda service. To evaluate the system performance, the memory assigned to the lambda function, and the number of

users access the system at once and the number of requests each user raised parameters is changed.

Values are measured using the following combinations.

1. 5 concurrent users sending 20 requests each user
2. 10 concurrent users sending 20 requests each user
3. 20 concurrent users sending 20 requests each user

Solution service is provided with memory with following values.

1. 128 MB
2. 256 MB
3. 512 MB
4. 1024 MB

The commonalities of the response time is measured with the time consumed by 95% of the requests and the 99% of requests. For each category those are illustrated in a different chart.

5.4.1 Five concurrent users sending 20 requests each user

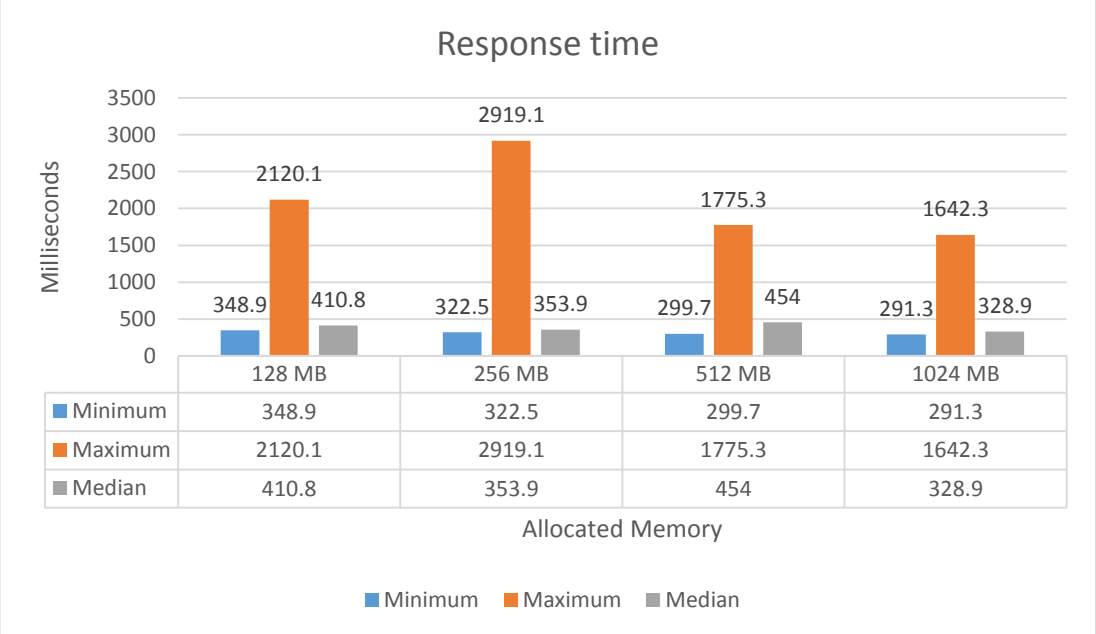

Figure5.14: Response time against the allocated memory

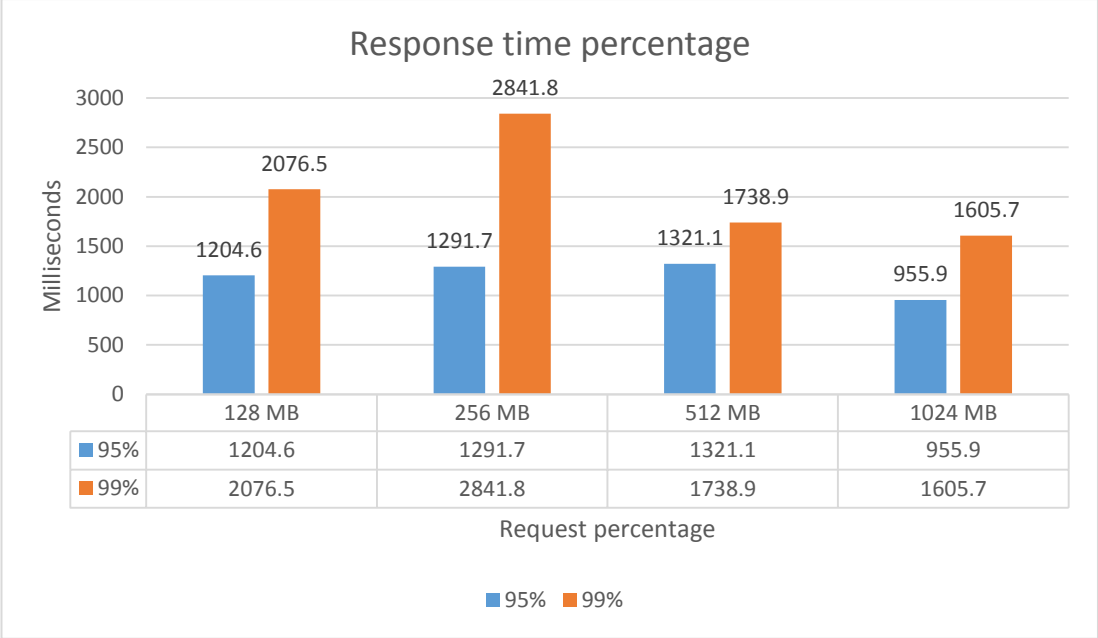

Figure5.15: Response time percentage consuming time

5.4.2 Ten concurrent users sending 20 requests each user

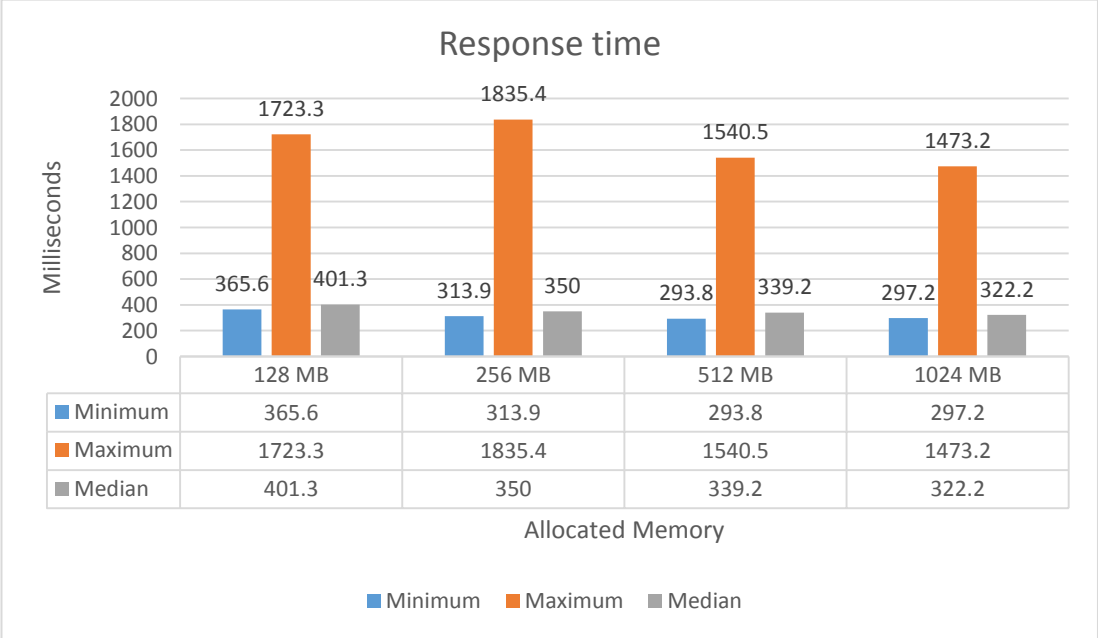

Figure5.16: Response time against the allocated memory

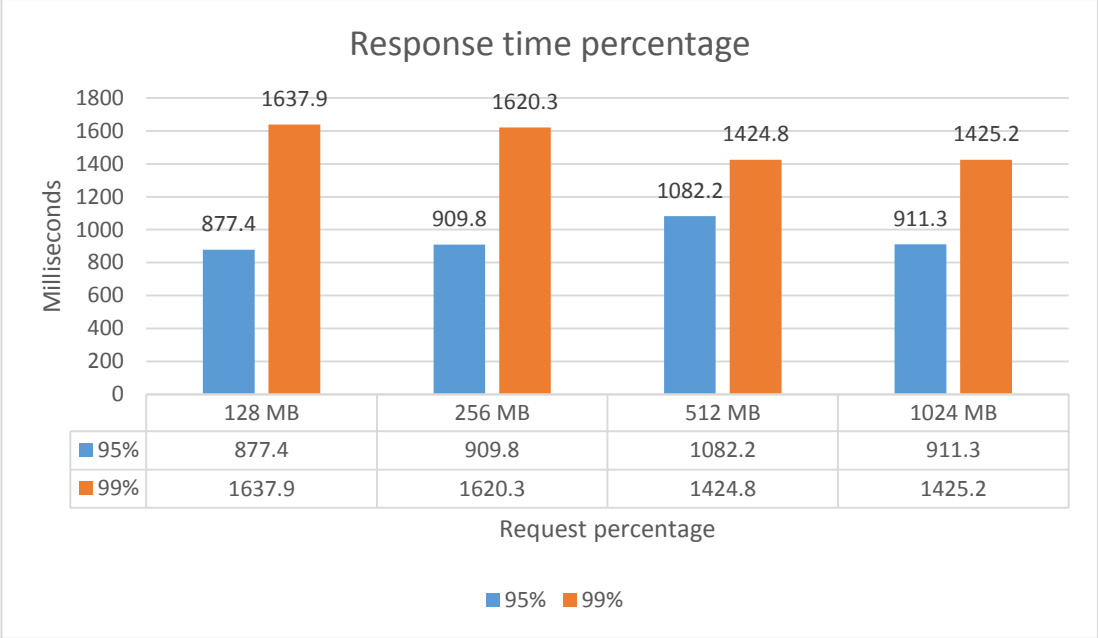

Figure5.17: Response time percentage consuming time

5.4.3 Twenty concurrent users sending 20 requests each user

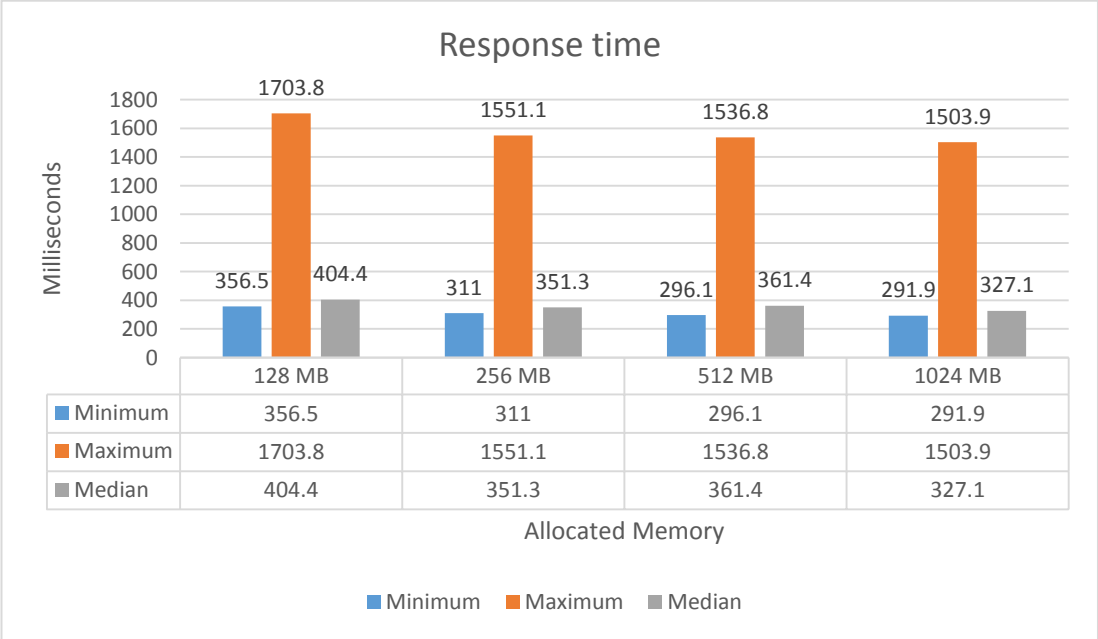

Figure5.18: Response time against the allocated memory

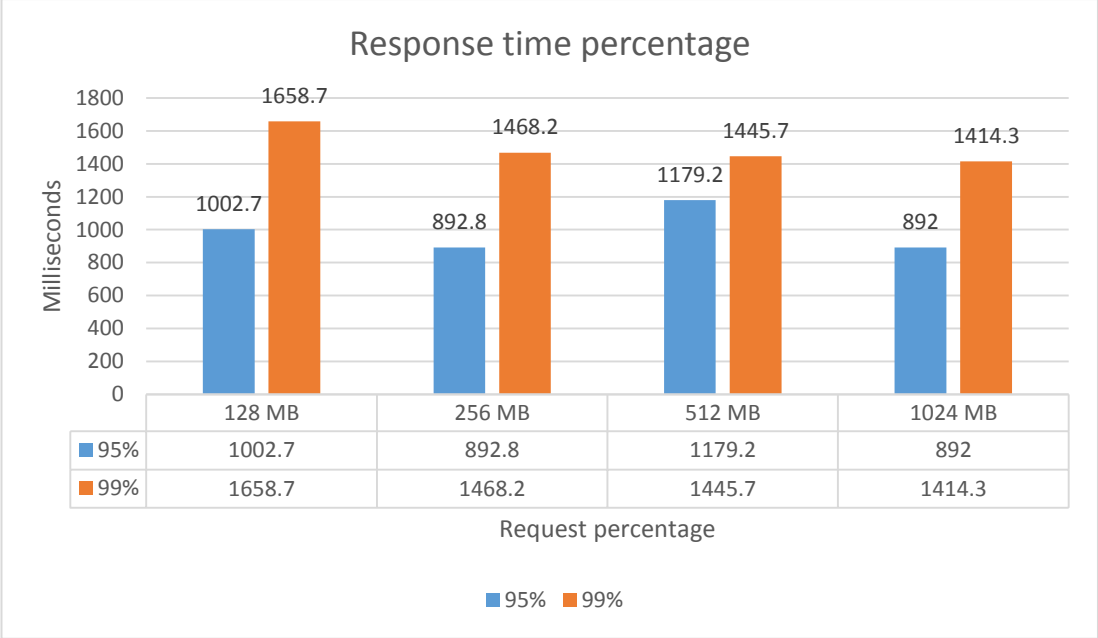

Figure5.19: Response time percentage consuming time

When we evaluate the system against the load, we applied described in the above scenarios and we can observe the following.

Throughout the different loads applied to the system, the system can scale and provide the service without a drastic deviation of the values. That means the solution is capable of handling different user loads, parallel users, and serve. It increases the availability and scalability of the solution.

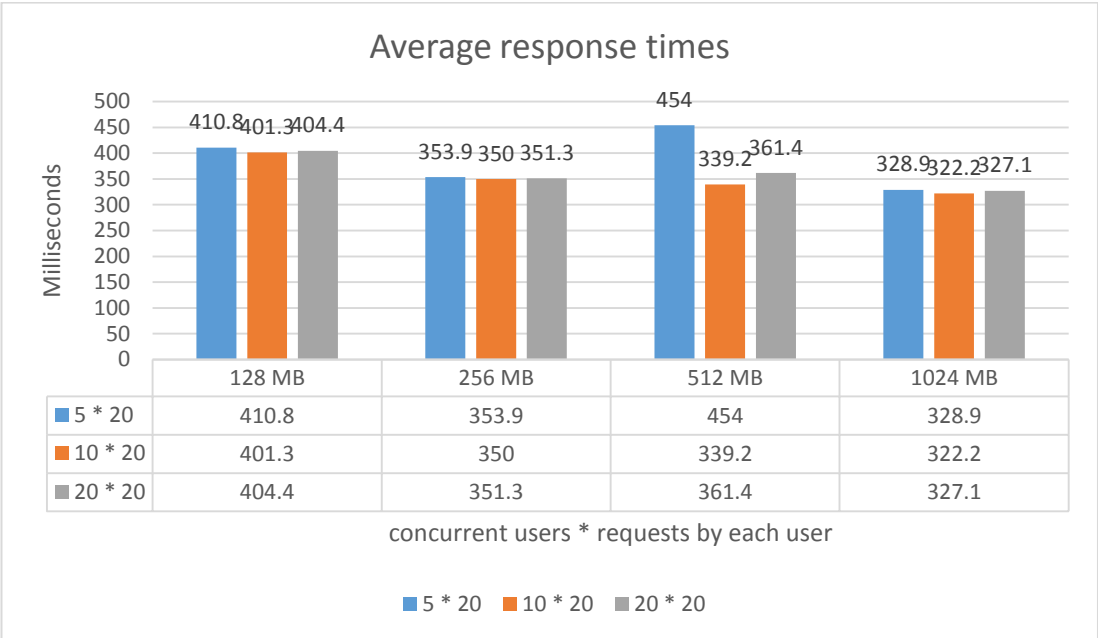

Figure5.20: Average response time against memory

5.5 Evaluation from industry experts

Another round of evaluation happened in this solution with the industry experts. A class diagram is shared with the industry experts and allowed them to add comments on them. Following class diagram is shared between senior personal in the IT industry who work as software developers more than 7 years.

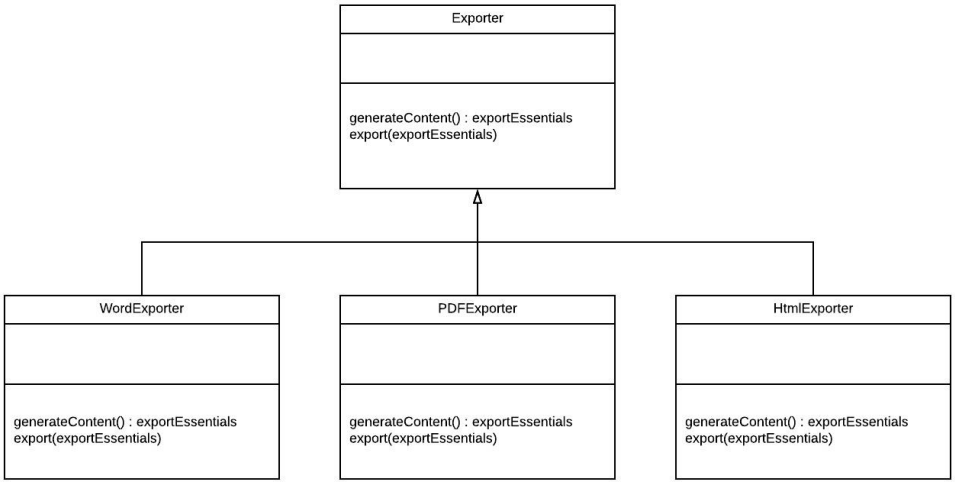

Figure5.21: Sample Class Diagram

For this evaluation, two industry professionals have selected who holds the designation Associate Technical Specialist. Professionals with this designation involve highly in designing of the software solutions and the implementation of the solutions. Compared to the senior engineers these professionals are involved in solution designing, as well as implementation. Compared to the Architects, they involve in actual implementations. From the designing implementing and delivering ends, Technical Specialists own the whole ownership.

I selected the following industry professionals in this evaluation.

1. Expert 1 – Associate Technical Specialist with 9 years of Industry experience
    a. This diagram completes the Single Responsibility rule and the Open Close principle according to the way different responsibilities are assigned to the different classes. Development is more structured when responsibilities are assigned like this and it will contribute to the maintenance phase as well.
2. Expert 2 – Associate Technical Specialist with 8 years of Industry experience

a. In the parent class generateContent() and export(exportEssentials) methods better to be abstract methods rather than overriding them in the child classes. Access modifiers should be displayed properly in the class diagram which is not present at the moment.

Following is the output we receive from the solution service.


```
[
  {
    "_message": "Following classes are not associate with abstract layers. WordExporter PDFExporter HtmlExporter"
  },
  {
    "_message": " Try to use Composition over inheritance in order to make the design more flexible."
  },
  {
    "_message": "generateContent and export are aspects that vary. So they can be seperated."
  }
]
```


Figure5.22: Result for the class diagram evaluation

The first message it says associate the child classes with abstract layers. One approach is just as Thilina proposed, adding abstract methods of generateContent() and export(exportEssentials) methods. Having abstract layers in between the parent class and child classes here is tricky since this diagram satisfies the inheritance.

In the message it says to use composition over inheritance. The reason is in the development phase both inheritance and composition help to structure the code well with divided responsibilities, however, in the maintenance mode, the Composition is more change friendly than the inheritance. When the first message and the second message are combined, we can derive that instead of inheritance we better structure this diagram with composition.

The third message says to identify the aspects that vary and separate and allow them to vary as per their desire. This allows us to predict the areas of a business entity that tend to change in the maintenance mode. Including this message with the first one, those aspects can add through an abstraction layer either an interface or an abstract

class. The resulted class diagram can be derived as follows.

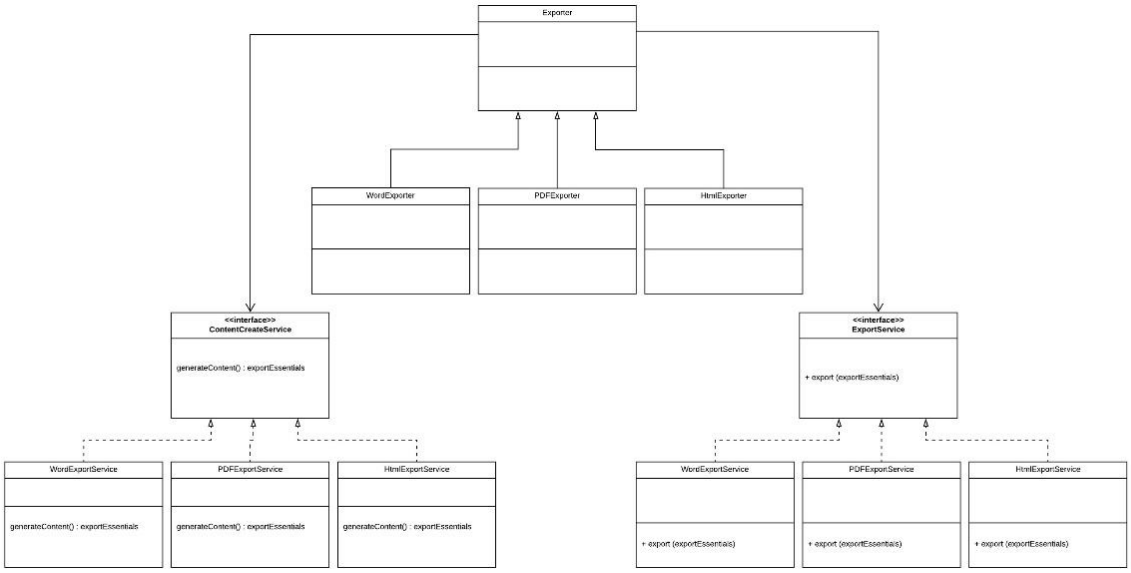

Figure5.23: Extended class diagram after the evaluation results

Chapter 06

Conclusion

Conclusion

Throughout this thesis, the Defect Prediction and Model-Driven Engineering are discovered. Importance of Defect Prediction is very high since defects consume a lot of resources in terms of time, cost, and human resources. The defects catch in the later parts of the software development life cycle are too much costly. Therefore, the Defect Prediction Model in whatever a viewpoint with higher accuracy is very much important.

Model-Driven Engineering also provides a well-structured mechanism in Software Development. The distance between the problem and the solution is better explained using the Models. Well-structured development always saves costs. Therefore, the focus on this thesis to combine those two for enhancing an efficient Software Development effort.

Defect Prediction Models were examined in different viewpoints. There is a trend to use data science algorithms and techniques for the use of defect prediction. Legacy systems with enough existing project data match the models that use the data science algorithms. However new projects that have less data, there are some problems to use data science technologies.

In this solution explained in the thesis, it tries to ensure or highlight the defect preventing possibilities of the Design Principles. Software product development and maintenance modes, the Design patterns valuable in preventing the defects while solving customer problems. Those Design Principles are the pillars of the Design Patterns as well. Design principles lets the developers to design the code in a way that requirement changes can be absorbed into the source code. The agile software development method is very popular today because it allows to absorb the software changes in an acceptable manner. However Design Principle allows you to make the source code is ready to be agile. There are Software Engineering best practices associated with agile development such as pair programming, unit testing, code refactoring etc. However for any of those practices good design backbone needs to be there to act as the pillar and the discipline of the source code.

The solution of this thesis is an external service that can be used with different architectures. Since Service Oriented Architecture playing a major role in today Software development, exposing this as a service will help to get the best out of this.

This service will accept the designs in a way that it can digest which is a JSON format. JSON Schema is a popular data expression mechanism today. The simplicity of using JSON format powers this solution and JSON is not a strange thing nowadays. Therefore JSON schema-based JSON format can be embedded in any kind of platform that can be facilitated with the solution explained in this thesis.

This solution can be extended towards a class diagram modeling guide lining solution. When a developer starts to work with a class diagram, this solution can provide recommendations on the approach developer can follow to prevent defects that can arise during the development phase, and the maintenance and requirements change phases. These recommendations will help to prevent the source code from being fragile against the software changes applied.

The current solution is targeting the UML class diagram evaluation. Model-Driven Engineering is having several other model types that can be used as a problem solution linking method. One limitation of this solution is focusing on UML class diagrams only. Defects of other UML models like activity, sequence, component diagrams are not tracking in this solution can be defined as a limitation in this solution.

This thesis expects to guide Software development with a reduced impact from defects, mainly in the Software maintenance phase.

Reference List

[1] Marco Brambilla, Jordi Cabot, Manuel Wimmer. (2012) Model-Driven Software Engineering in Practise. Morgan & Claypool Publishers.

[2] Robert France, Bernard Rumpe, Model Driven Development of Complex Software: A Research RoadMap. Future of Software Engineering 2007.

[3] A brief introduction to model-driven engineering. May 24th 2020. http://www.scielo.org.co/scielo.php?script=sci_arttext&pid=S0123-921X2014000200011.

[4] Manifesto for Agile Software Development. Retrieved May 11th 2020. http://agileManifesto.org

[5] Qing He, Beijun Shen, Yuting Chen, Sotware Defect Prediction Using Semi – Supervised Learning with Change Burst Information. IEEE Computer Society 2016.

[6] Wenjing Han, Chung-Horng Lung, Samuel A. Ajila, Empirical Investigation of Code and Process Metrics for Defect Prediction, IEEE Computer Society 2016.

[7] Kanika Chandra, Gagan Kapoor, Rashi Kohli, Archana Gupta, IMPROVING SOFTWARE QUALITY USING MACHINE LEARNING, 1st International Conference on Innovation and Challenges in Cyber Security 2016.

[8] Ankita Bansal, Ruchika Malhotra, Kimaya Raje, Analyzing and Assessing the Security-Related Defects, 1st International Conference on Innovation and Challenges in Cyber Security. 2016.

[9] Puntitra Sawadpong, Toward a Defect Prediction Model of Exception Handling Method Call Structures, ACM SE ’14, March 28 - 29 2014.

[10] Feng Zhang, Ahmed E.Hassan, Shane McIntosh, Ying Zou, The use of Summation to Aggregate Software Metrics Hinders the Performance of Defect Prediction Models. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, 2016.

[11] Xiaobing Sun, Xiangyue Liu, Bin Li, Yucong Duan, Hui Yang, Jianjun Hu, Exploring Topic Models in Software Engineering Data Analysis: A Survay, IEEE 2016.

[12] Chun Shan, Hongjin Zhu, Changzhen Hu, Jin Cui, Jinefeng Xue, Software Defect Prediction Model Based on Improved LLE-SVM. 4th International Conference on Computer Science and Network Technology, 2015.

[13] Komukai-Toshiba-cho, Saiwai-ku, Kawasaki, Superposed Naïve Bayes for Accurate and Interpretable Prediction, IEEE 14th International Conference on Machine Learning and Applications, 2015.

[14] Carol Woody, Robert Ellison, William Nichols, Predicting Cybersecurity Using Quality Data, IEEE, 2015.

[15] Wutthichai Chansuwath, Twittie Senivongse, A Model-Driven Development of Web Applications Using AngularJS Framework. IEEE, 2016.

[16] OMG, UML Profile. [Online]. Available: http://www.omg.org/mda/specs.htm. Last Accessed: 30 Mar 2016.

[17] Eric Freeman, Elisabeth Freeman, Kathy Sierra and Bert Bates. Head First Design Patterns, 2004.

[18] MxGraph. Retrieved May 11th , 2020. https://github.com/jgraph/mxgraph

[19] Asadullah Shaikh, Overview of Slicing and Feedback Techniques for Efficient Verification of UML/OCL Class Diagrams.

[20] J.cabot, R Clariso, and D.Riera. Verification of uml/ocl class diagrams using constraint programming. In ASE 2007, pages 547-548. ACM 2007.

[21] The ECLiPSe Constraint Programming System. http://www.ecipseclp.org/, Mar 2007, version 5.10. Last checked on January 11, 2018.

[22] Object Management Group (OMG). Object Constraint Language. Retrieved May 11th, 2020. http://www.omg.org/spec/ocl.

[23] Object management Group (OMG). Unified Modelling Lanuguage homepage 2018. Retrieved May 11th 2020. http://www.omg.org.

[24] Shamsul Huda, Sultan Alyahya, MD Mohsin Ali, Shafiq Ahmed, Jemal Abawajy, Hmood Al Dossari, John Yearwood. A Framework for Software Defect Prediction and Metric selection.

[25] L. Pelayo and S.Dick, “Applying novel resampling strategies to software defect prediction” in Proc. Annu. Meeting North Amer. Fuzzy Inf. Process. Soc (NAFIPS), Jan. 2007, pp. 69-72.

[26] H.B Yadav and D. K Yadav, “A Fussy logic based approach for phase wise software defects prediction using software metrices.” Inf. Softw. Tech-nol., vol 63, pp. 44-57. Jul. 2015.

[27] T.Bages, “An essay towards solving a problem in the doctrine of changes.” Philos, Trans, Roy. Soc. London, no 53, pp. 370-418, 1763. [Online]. Available: http://www.sta.ucls.edu/history/essay.pdf

[28] N. Cristianini and J.Shawe-Taylor, An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods. New York, NY, USA: Cambridge Univ. Press 2000

[29] J.R. Quinlan, “Induction of decision trees,” Mach. Learn., vol. 1, no0 1, pp. 81-106, Mar. 1986.

[30] B. Krose and P.V.D Smagt, An Introduction toNeural Networks. Amsterdam, The Netherlands: Univ. Amsterdam, 1993.

[31] J. A. Rodger, “Toward reducing failure risk in an integrated vehicle health maintenance system: A fuzzy multi-sensor data fusion Kalman filter approach for IVHMS,” Expert Syst. Appl., vol. 39, no. 10, pp. 9821-9836, Aug. 2012.

[32] F. Aparisi and J.Sanz, “Interpreting the out-of-control signals of multi-variate control charts employing neural networks,”. Int. J. Comput., Electl, Autom., Control Inf. Eng., vol. 4, no. 1, pp. 24-28, 2010.

[33] A. A Asad and I. Alsmadi, “Evaluating the impact of software metrices on defects prediction. Part 2,” Comput. Sci. J. Moldova, vol. 22, no. 1, pp. 127-144, 2014.

[34] R. Malhotra, “A systematic review of machine learning techniques for software fault prediction,” Appl. Soft Comput., vol. 27, pp. 504-518, Feb. 2015.

[35] H. Zhang, “An investigation of the relationships between lines of code and the defects.” In Proc. IEEE Int. Conf. Softw. Maintenance (ICSM), Sep. 2009, pp. 274-283.

[36] L. Yu and A. Mishra, “Experience predicting fault-prone software modules using complexity metrices” Quality Technolol. Quantitative Manage., vol. 9, no. 4, pp. 421-433, May 2012. http://web.it.nctu.edu.tw/~qtqm/qtqmpaers/2012V9N4/2012V9N4/2012_F7.pdf

[37] Feng Zhang and Audris Mockus and Iman Keivanloo and Ying Zou, Towards Building a Universal Defect Prediction Model, 2014.

[38] J. Nam, S. J. Pan, and S. Kim. Transfer defect learning. In Proceedings of the 35th International Conference on Software Engineering, pages 382–391, 2013

[39] F. Zhang, A. Mockus, Y. Zou, F. Khomh, and A. E.Hassan. How does context affect the distribution of software maintainability metrics? In Proceedings of the 29th IEEE International Conference on Software Maintainability, pages 350 – 359, 2013.

[40] N. Ohlsson and H. Alberg, ‘‘Predicting fault-prone software modules in telephone switches,’’ IEEE Trans. Softw. Eng., vol. 22, no. 12,pp. 886–894, Dec. 1996

[41] R. Malhotra, ‘‘A systematic review of machine learning techniques for software fault prediction,’’ Appl. Soft Comput., vol. 27, pp. 504–518, Feb. 2015.

[42] S. R. Chidamber and C. F. Kemerer, ‘‘A metrics suite for object oriented design,’’ IEEE Trans. Softw. Eng., vol. 20, no. 6, pp. 476–493, Jun. 1994.

[43] T. T. Nguyen, T. N. Nguyen, and T. M. Phuong. Topic-based defect prediction (nier track). In Proceedings of the 33rd International Conference on Software Engineering, ICSE ’11, pages 932–935, New York, NY, USA, 2011. ACM.

[44] D. Andrzejewski, A. Mulhern, B. Liblit, and X. Zhu.Statistical debugging using latent topic models. In Proceedings ofthe 18th European Conference on Machine Learning, pages 6–17, 2007

[45] T.-H. Chen, S. W. Thomas, M. Nagappan, and A. E.Hassan. Explaining software defects using topic models. In Proceedings of the 9th IEEE Working Conference on Mining Software Repositories, pages 189–198, 2012.

[46] W. Hu and K. Wong. Using citation influence to predict software defects. In Proceedings of the 10th Working Conference on Mining Software Repositories, MSR ’13, pages 419–428, Piscataway, NJ, USA, 2013. IEEE Press.

[47] P. Sawadpong, E. B. Allen, and B. J. Williams. Exception handling defects: An empirical study. In IEEE 14th International Symposium on High-Assurance Systems Engineering, pages 90–97, 2012

[48] B. G. Ryder. Constructing the call graph of a program. IEEE Transactions on Software Engineering, SE-5(3):216–226, May 1979.

[49] C. Marinescu. Are the classes that use exceptions defect prone? In Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIMWorkshop on Software Evolution, pages 56–60. ACM, 2011.

[50] L. Briand, S. Morasca, and V. Basili. Property-based software engineering measurement. IEEE Transactions on Software Engineering, 22(1):68–86, 1996

[51] T. M. Khoshgoftaar and E. B. Allen. Logistic regression modeling of software quality. International Journal of Reliability, Quality & Safety Engineering, 6(4):303–317, 1999

[52] Yucong Duan, Qiang Duan, Ruisheng Shi, Honghao Gao, Toward an Automated View Abstraction for Distributed Model-Driven Service Development. IEEE International Conference on Services Computing, 2016

Appendices

Appendix A - Sample Json Payload.


```
{
	"ModelType":"ClassDiagram",
	"ModelElements":[
		{
			"id":1,
			"type":"class",
			"elementName": "Exporter",
			"parentElements":null,
			"isAbstractClass":"false",
			"childElements":[
{"childId":2,
"relationType":"Inheritance"},
{"childId":3,
"relationType":"Inheritance"},
{"childId":4,
"relationType":"Inheritance"}
],
			"attributes":null,
			"behaviors":[
{"name":"generateContent",
"returnType":"ExportEssentials", "parametersInOrder":null,
"isOverriden":false, "isAbstract": false},
{"name":"export",
"returnType":"void", "parametersInOrder":"exportEssentials",
"isOverriden":false, "isAbstract": false}

]
		},
		{
			"id":2,
			"type":"class",
			"elementName": "WordExporter",
			"parentElements":[{"parentId":1,
"relationType":"Inheritance"}],
			"childElements":null,
			"attributes":null,
			"behaviors":[
{"name":"generateContent",
"returnType":"ExportEssentials", "parametersInOrder":null,
"isOverriden":true},
{"name":"export",
"returnType":"void", "parametersInOrder":null, "isOverriden":true}
]
		},
		{
			"id":3,
			"type":"class",
			"elementName": "PDFExporter",
			"parentElements":[{"parentId":1,
"relationType":"Ihnheritance"}],
			"childElements":null,
			"attributes":null,
			"behaviors":[
```

```
                                                {"name":"generateContent",
"returnType":"ExportEssentials", "parametersInOrder":null,
"isOverriden":true},
                                                {"name":"export",
"returnType":"void", "parametersInOrder":null, "isOverriden":true}
                                                ]
                },
                {
                        "id":4,
                        "type":"class",
                        "elementName": "HtmlExporter",
                        "parentElements":[{"parentId":1,
"relationType":"Inheritance"}],
                        "childElements":null,
                        "attributes":null,
                        "behaviors":[
                                                {"name":"generateContent",
"returnType":"ExportEssentials", "parametersInOrder":null,
"isOverriden":true},
                                                {"name":"export",
"returnType":"void", "parametersInOrder":null, "isOverriden":true}
                                                ]
                }
        ]
    }
```

Appendix B – Json message for a composition relation


```
{
	"ModelType":"ClassDiagram",
	"ModelElements":[
		{
			"id":1,
			"type":"class",
			"elementName": "Student",
			"parentElements":null,
			"isAbstractClass":"false",
			"childElements":[
{"childId":2,
"relationType":"Composition"}
],
			"attributes":[
{"name":"courseMaterial",
"type":"CourseMaterial"}
],
			"behaviors":[

]
		},
		{
			"id":2,
			"type":"class",
			"elementName": "CourseMaterial",
			"parentElements":[{"parentId":1,
"relationType":"Composition"}],
			"childElements":null,
			"attributes":null,
			"behaviors":[

]
		}
	]
}
```

Appendix C – Json message for an Aggregation relation


```
{
	"ModelType":"ClassDiagram",
	"ModelElements":[
		{
			"id":1,
			"type":"class",
			"elementName": "College",
			"parentElements":null,
			"isAbstractClass":"false",
			"childElements":[
{"childId":2,
"relationType":"Aggregation"}
],
			"attributes":[
{"name":"swimmingPool",
"type":"SwimmingPool"}
],
			"behaviors":[

]
		},
		{
			"id":2,
			"type":"class",
			"elementName": "SwimmingPool",
			"parentElements":[{"parentId":1,
"relationType":"Aggregation"}],
			"childElements":null,
			"attributes":null,
			"behaviors":[

]
		}
	]
}
```

Appendix D - Main Json Schema

```
{
   "definitions": {},
   "$schema": "http://json-schema.org/draft-06/schema#",
   "$id": "http://example.com/example.json",
   "type": "object",
   "properties": {
     "ModelType": {
       "$id": "/properties/ModelType",
       "type": "string",
       "title": "The Modeltype Schema",
       "description": "This explains the diagram type. Sofar type
Classdiagram supports.",
       "default": "",
       "examples": [
         "ClassDiagram"
       ]
     },
     "ModelElements": {
       "$id": "/properties/ModelElements",
       "type": "array",
       "items": {
         "$id": "/properties/ModelElements/items",
         "type": "object",
         "properties": {
           "id": {
             "$id": "/properties/ModelElements/items/properties/id",
             "type": "integer",
             "title": "The Id Schema",
             "description": "This is the ID of the ModelElemnet.This uses
to uniquely identify the ModelElement.",
             "default": 0,
             "examples": [
               1
             ]
           },
           "type": {
             "$id": "/properties/ModelElements/items/properties/type",
             "type": "string",
             "title": "The Type Schema",
             "description": "This is the ModelELement type. This can be
class, interface, abstractClass",
             "default": "class",
             "examples": [
               "class",
               "interface",
                           "abstractClass"
             ]
           },
           "elementName": {
             "$id":
"/properties/ModelElements/items/properties/elementName",
             "type": "string",
             "title": "The Elementname Schema",
             "description": "Name of the Element."
           },
           "parentElements": {
```

```
            "$id":
"/properties/ModelElements/items/properties/parentElements",
            "type": "null",
            "title": "The Parentelements Schema",
            "description": "This is the list of parent model elements
that are associated with current model element.",
            "default": null,
            "examples": [
              null
            ]
          },
          "isAbstractClass": {
            "$id":
"/properties/ModelElements/items/properties/isAbstractClass",
            "type": "string",
            "title": "The Isabstractclass Schema",
            "description": "This indicates current model element is an
abstract class or not.",
            "default": "false",
            "examples": [
              "false"
            ]
          },
          "childElements": {
            "$id":
"/properties/ModelElements/items/properties/childElements",
            "type": "array",
            "items": {
              "$id":
"/properties/ModelElements/items/properties/childElements/items",
              "type": "object",
              "properties": {
                "childId": {
                  "$id":
"/properties/ModelElements/items/properties/childElements/items/properties
/childId",
                  "type": "integer",
                  "title": "The Childid Schema",
                  "description": "This explains the child model elements
associated with current model element.",
                  "default": 0,
                  "examples": [
                    2
                  ]
                },
                "relationType": {
                  "$id":
"/properties/ModelElements/items/properties/childElements/items/properties
/relationType",
                  "type": "string",
                  "title": "The Relationtype Schema",
                  "description": "An explanation about the purpose of
this instance.",
                  "default": "",
                  "examples": [
                    "Inheritance"
                  ]
                }
              }
```

```
            }
          },
          "attributes": {
            "$id":
"/properties/ModelElements/items/properties/attributes",
            "type": "null",
            "title": "The Attributes Schema",
            "description": "An explanation about the purpose of this
instance.",
            "default": null,
            "examples": [
              null
            ]
          },
          "behaviors": {
            "$id":
"/properties/ModelElements/items/properties/behaviors",
            "type": "array",
            "items": {
              "$id":
"/properties/ModelElements/items/properties/behaviors/items",
              "type": "object",
              "properties": {
                "name": {
                  "$id":
"/properties/ModelElements/items/properties/behaviors/items/properties/nam
e",
                  "type": "string",
                  "title": "The Name Schema",
                  "description": "These are the methods or the behaviors
of the class.",
                  "default": "",
                  "examples": [
                    "quack"
                  ]
                },
                "returnType": {
                  "$id":
"/properties/ModelElements/items/properties/behaviors/items/properties/ret
urnType",
                  "type": "string",
                  "title": "The Returntype Schema",
                  "description": "The return type of the method",
                  "default": "",
                  "examples": [
                    "void"
                  ]
                },
                "parametersInOrder": {
                  "$id":
"/properties/ModelElements/items/properties/behaviors/items/properties/par
ametersInOrder",
                  "type": "null",
                  "title": "The Parametersinorder Schema",
                  "description": "Method parameters",
                  "default": null,
                  "examples": [
                    null
                  ]
```

```
                },
                "isOverriden": {
                  "$id":
"/properties/ModelElements/items/properties/behaviors/items/properties/isO
verriden",
                  "type": "boolean",
                  "title": "The Isoverriden Schema",
                  "description": "An overriden method or not",
                  "default": false,
                  "examples": [
                    false
                  ]
                },
                "isAbstract": {
                  "$id":
"/properties/ModelElements/items/properties/behaviors/items/properties/isA
bstract",
                  "type": "boolean",
                  "title": "The Isabstract Schema",
                  "description": "An abstract method or not",
                  "default": false,
                  "examples": [
                    false
                  ]
                }
              }
            }
          }
        }
      }
    }
  }
      }
```

Appendix E – Source Code of the solution – server.js

```
'use strict'

const express = require( "express" );
const bodyParser = require('body-parser')

const Evaluator = require('./src/components/Evaluator')


const app = express();
const port = 8080; // default port to listen

app.use(bodyParser.json())

app.post( "/api/v1/classmodel/", ( req, res ) => {
    let evaluator = new Evaluator();
    res.send( evaluator.evaluateData(req) );
} );


// start the Express server
app.listen( port, () => {
    console.log( `server started at http://localhost:${ port }` );
});
```

Appendix F – Source Code of the solution – Evaluator.js

```
module.exports = class EvaluatorClass {

    evaluateData(data)
    {
        let classDiagramData = data.body;
        return
this.identifyDesignPrincipleViolations(classDiagramData.ModelElements)
    }

    identifyDesignPrincipleViolations(classData)
    {
        const InheritanceText = 'Inheritance';
        const ClassText = 'class';
        const CompositionText = 'Composition';
        const AssociationText = 'Association';
        const AggregationText = 'Aggregation';
        const coupingPredictions = [];
        classData.forEach((individualClass) => {
            if (individualClass.parentElements && individualClass.type ==
ClassText) {
                coupingPredictions.push(this.getMessageObject('Class ' +
individualClass.elementName + ' has no dependancy with an abstract layer.
Possible violation of Design Principle'))
            }

            if (individualClass.childElements) {
                individualClass.childElements.forEach((childElementClass)
=> {
                    if (childElementClass.relationType &&
childElementClass.relationType === InheritanceText) {
                        const childClass = this.getClassFromId(classData,
childElementClass.childId);

coupingPredictions.push(this.getMessageObject('Class ' +
individualClass.elementName + ' has a inheritance dependancy with class '
+ childClass.elementName + '. Behavior/Method implentation may vary in the
class child ' + childClass.elementName))
                    }
                });
            }

            if (individualClass.behaviors &&
individualClass.childElements) {
                let hasInheritedChild = false;
                individualClass.childElements.forEach((childElementClass)
=> {
                    if (childElementClass.relationType &&
childElementClass.relationType === InheritanceText) {
                        hasInheritedChild = true;
                    }
                });

                if (hasInheritedChild === true) {
                    individualClass.behaviors.forEach((behavior) => {

coupingPredictions.push(this.getMessageObject('Class ' +
individualClass.elementName + ' has a behavior / Method ' + behavior.name
```

```
+ ' that can have deviated behaviors in the child classes. Better to refer
it through an abstraction layer'))
                    });
                }
            }

            if (individualClass.childElements) {
                individualClass.childElements.forEach((childElementClass)
=> {
                    if (childElementClass.relationType &&
childElementClass.relationType == CompositionText) {
                        const childClass = this.getClassFromId(classData,
childElementClass.childId);
                        if (childClass.type == ClassText) {

coupingPredictions.push(this.getMessageObject('Class ' +
individualClass.elementName + ' has a Composition relationship with Class
' + childClass.elementName + ' without an abstraction layer'))
                        }
                    }
                });
            }

            if (individualClass.childElements) {
                individualClass.childElements.forEach((childElementClass)
=> {
                    if (childElementClass.relationType &&
childElementClass.relationType == AssociationText) {
                        const childClass = this.getClassFromId(classData,
childElementClass.childId);
                        if (childClass.type == ClassText) {

coupingPredictions.push(this.getMessageObject('Class ' +
individualClass.elementName + ' has an Association relationship with Class
' + childClass.elementName + ' without an abstraction layer'))
                        }
                    }
                });
            }

            if (individualClass.childElements) {
                individualClass.childElements.forEach((childElementClass)
=> {
                    if (childElementClass.relationType &&
childElementClass.relationType == AggregationText) {
                        const childClass = this.getClassFromId(classData,
childElementClass.childId);
                        if (childClass.type == ClassText) {

coupingPredictions.push(this.getMessageObject('Class ' +
individualClass.elementName + 'has an Aggregation relationship with Class
' + childClass.elementName + ' without an abstraction layer'))
                        }
                    }
                });
            }

        });
```

```
        return coupingPredictions;
    }

    getClassFromId(data, id)
    {
        let returnObj;
        data.forEach((className) => {
            if (className.id == id) {
                returnObj = className;
                return className;
            }
        })

        return returnObj;
    }

    getMessageObject(message)
    {
        return {
            _message: message
        };
    }
}
```