

TOWARDS THE DEFINITION OF SOFTWARE MODEL

STABILITY METRICS

BY

AMJAD MAJED AHMAD ABU HASSAN

A Thesis Presented to the
DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

SOFTWARE ENGINEERING

MAY 2015

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN- 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **Amjad Majed Ahmad Abu Hassan** under the direction his thesis advisor and approved by his thesis committee, has been presented and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN SOFTWARE ENGINEERING.**



Dr. Mohammad Alshayeb
(Advisor)



Dr. Abdulaziz Alkhoraidly
Department Chairman



Dr. Mahmood Niazi
(Member)



Dr. Salam A. Zummo
Dean of Graduate Studies



Dr. Sajjad Mahmood
(Member)

1/6/15
Date



© Amjad Abu Hassan

2015

Dedication

To my Parents

For their LOVE

ACKNOWLEDGMENTS

In the name of Allah, the Most Gracious, the Most Merciful

First and foremost, Alhamdulillah All praise to Almighty Allah, who gave me the power to accomplish my master's degree.

I acknowledge King Fahd University of Petroleum & Minerals for supporting this research.

All appreciation to my advisor; Dr. Mohammad Alshayeb, who helped me and encouraged me during my thesis journey; he was a teacher, a friend, and a brother. I wish to thank my dissertation committee members, Dr. Mahmood Niazi, and Dr. Sajjad Mahmood, for their help and support.

Finally, I wish to express my gratitude to my family members for their prayers and patience. I would also like to thank all my KFUPM colleagues, who provided me the encouragement in dealing with difficult times during the thesis journey.

TABLE OF CONTENTS

ACKNOWLEDGMENTS	V
TABLE OF CONTENTS	VI
LIST OF TABLES.....	XI
LIST OF FIGURES.....	XIII
LIST OF ABBREVIATIONS.....	XVI
ABSTRACT.....	XVIII
ملخص الرسالة	XIX
CHAPTER 1 INTRODUCTION.....	1
1.1 Motivation	2
1.2 Research Objectives.....	3
1.3 Contributions	3
1.4 The Outline	4
CHAPTER 2 BACKGROUND	5
2.1 Unified Modeling Language (UML).....	5
2.1.1 Structural View.....	6
2.1.2 Behavioral View	7
2.1.3 Functional View.....	8
2.2 UML Class Diagram	8
2.2.1 Classes.....	9
2.2.2 Class Diagram Relationships	9
2.2.3 Association classes	11

2.3	UML Use Case Diagram	11
2.3.1	Use Cases	12
2.3.2	Actors	12
2.3.3	System Boundaries	12
2.3.4	Use Case Diagram Relationships	12
2.4	UML Sequence Diagram	13
2.4.1	Participant	14
2.4.2	Time	14
2.4.3	Messages	14
2.4.4	Notes	15
2.4.5	Messages Type	15
CHAPTER 3 LITERATURE REVIEW		17
3.1	Stability	17
3.1.1	Architecture Level Metrics	17
3.1.2	Class Level Metrics	20
3.1.3	System Level Metrics	23
3.2	Similarity	26
3.3	Summary	28
CHAPTER 4 RESEARCH METHODOLOGY		33
4.1	Analysis	35
4.2	Evaluation	36
4.3	Propose the Metrics	36
CHAPTER 5 STRUCTURAL STABILITY		37
5.1	Assessment	37

5.1.1	Classifiers	38
5.1.2	Comments	39
5.1.3	Packages.....	39
5.1.4	Dependency Relationship.....	40
5.1.5	Association Relationship	43
5.1.6	Aggregation Relationship	46
5.1.7	Composition Relationship.....	49
5.1.8	Inheritance Relationship	52
5.1.9	Realization Relationship	54
5.1.10	Association classes.....	57
5.1.11	The Selected UML Class Diagram Identifier	59
5.1.12	Summary	59
5.2	Terminology and Formalism.....	60
5.3	Structural Stability Metric.....	61
5.3.1	Example.....	65
CHAPTER 6 FUNCTIONAL STABILITY		68
6.1	Assessment	68
6.1.1	Actor	68
6.1.2	Use Case	69
6.1.3	System Boundaries	71
6.1.4	Actor Relationships	71
6.1.5	Generalization Relationship	74
6.1.6	Include Relationship	76
6.1.7	Extend Relationship.....	78
6.1.8	The Selected UML Use Case Diagram Identifier	80

6.1.9	Summary	81
6.2	Terminology and Formalism.....	81
6.3	Functional Stability Metric	83
6.3.1	Example.....	88
CHAPTER 7 BEHAVIORAL STABILITY		91
7.1	Assessment	91
7.1.1	Participant.....	91
7.1.2	Stereotypes	93
7.1.3	Messages.....	94
7.1.4	A synchronous message	96
7.1.5	An asynchronous message.....	97
7.1.6	A return message	98
7.1.7	Creation Message & Destruction Message	99
7.1.8	Notes, Activation Bars, and Actors	101
7.1.9	Time	101
7.1.10	The Selected UML Sequence Diagram Identifier.....	102
7.2	Terminology and Formalism.....	104
7.3	Behavioral Stability Metric.....	106
7.3.1	Example.....	111
CHAPTER 8 THEORETICAL VALIDATION		113
8.1	Structural Stability Metric Validation	116
8.2	Functional Stability Metric Validation	118
8.3	Behavioral Stability Metric Validation.....	120
CHAPTER 9 CASE STUDIES		123

9.1	Case Study 1: ATM	123
9.2	Case Study 2: SCM.....	141
9.3	Case Study 3: ORA.....	151
9.4	Case Study 4: O-RED System	158
9.5	Case Study 5: HOSS System	163
9.6	Case Study 6: ESAP System.....	170
CHAPTER 10 CONCLUSION AND FUTURE WORK.....		176
10.1	Conclusion and Thesis Contribution.....	176
10.2	Future work.....	177
10.3	Threats to Validity	178
REFERENCES.....		179
VITAE.....		183

LIST OF TABLES

<i>Table 1 Stability Survey Summary</i>	29
<i>Table 2 Similarity Techniques Summary</i>	32
<i>Table 3 Dependency Relationship Possible Changes</i>	42
<i>Table 4 Association Relationship Possible Changes</i>	45
<i>Table 5 Aggregation Relationship Possible Changes</i>	48
<i>Table 6 Composition Relationship Possible Changes</i>	50
<i>Table 7 Inheritance Relationship Possible Changes</i>	53
<i>Table 8 Realization Relationship Possible Changes</i>	56
<i>Table 9 Association Class Possible Changes</i>	58
<i>Table 10 Each Classifier Properties</i>	66
<i>Table 11 Changes From version i to version i+1</i>	67
<i>Table 12 Possible Use Case Changes</i>	70
<i>Table 13 Possible Actor-Use case Relationships Changes</i>	73
<i>Table 14 Possible Actor-Actor Relationships Changes</i>	74
<i>Table 15 Generalization Relationship Possible changes</i>	75
<i>Table 16 Include Relationship Possible changes</i>	77
<i>Table 17 Extend Relationship Possible changes</i>	80
<i>Table 18 Use Case Sample Diagrams Properties</i>	88
<i>Table 19 Use Case Sample Changes From version i to version i+1</i>	89
<i>Table 20 Possible Identifier Changes</i>	104
<i>Table 21 All Messages Property for the Sequence Sample Diagrams</i>	111
<i>Table 22 Changes From version 1 to version 2 in Sequence Sample Diagrams</i>	112
<i>Table 23 ATM Case Study Summary</i>	124
<i>Table 24 ATM Class Diagrams Comparison</i>	127
<i>Table 25 ATM Class Diagram Comparison Results</i>	129
<i>Table 26 ATMStartUp Sequence Diagrams Comparison</i>	133
<i>Table 27 ATMStartUp Sequence Diagrams Comparison Results</i>	134
<i>Table 28 Deposit Sequence Diagrams Comparison</i>	136
<i>Table 29 Deposit Sequence Diagrams Comparison Results</i>	137
<i>Table 30 Withdrawal Sequence Diagrams Comparison</i>	139
<i>Table 31 Withdrawal Sequence Diagrams Results</i>	140
<i>Table 32 SCM Case Study Summary</i>	141
<i>Table 33 SCM Class Diagrams Comparison</i>	143
<i>Table 34 SCM Class Diagrams Comparison Results</i>	144
<i>Table 35 Purchase Sequence Diagram Comparison</i>	146
<i>Table 36 Purchase Sequence Diagram Comparison Results</i>	146

<i>Table 37 Replenish Sequence Diagram Comparison</i>	148
<i>Table 38 Replenish Sequence Diagram Comparison Results</i>	148
<i>Table 39 Source Sequence Diagram Comparison</i>	150
<i>Table 40 Source Sequence Diagram Comparison Results</i>	150
<i>Table 41 ORA Use Case Diagram Comparison</i>	154
<i>Table 42 ORA Class Diagram Comparison Results</i>	156
<i>Table 43 O-RED Class Diagram Comparison</i>	161
<i>Table 44 O-RED Class Diagram Comparison Results</i>	162
<i>Table 45 HOSS Use Case Diagram Comparison</i>	166
<i>Table 46 HOSS Use Case Diagram Comparison Results</i>	168
<i>Table 47 ESAP Use Case Diagram Comparison</i>	173
<i>Table 48 ESAP Use Case Diagram Comparison Results</i>	175

LIST OF FIGURES

<i>Figure 1 Hierarchical Classification of UML Diagrams [6]</i>	6
<i>Figure 2 Class Diagram Relationships</i>	11
<i>Figure 3 Use Case Diagram Relationships</i>	13
<i>Figure 4 Sequence Diagram Sample</i>	14
<i>Figure 5 Research Methodology</i>	34
<i>Figure 6 Class Diagram Sample</i>	37
<i>Figure 7 Detailed version of the class in class diagram</i>	38
<i>Figure 8 Four different ways of showing a class using UML notation</i>	39
<i>Figure 9 Class Diagram Dependency Relationship</i>	40
<i>Figure 10 Sample Code of Dependency Relationship</i>	41
<i>Figure 11 Class Diagram Association Relationship</i>	44
<i>Figure 12 Sample Code of Association Relationship</i>	44
<i>Figure 13 Class Diagram Aggregation Relationship</i>	47
<i>Figure 14 Sample Code of Aggregation Relationship</i>	47
<i>Figure 15 Class Diagram Composition Relationship</i>	49
<i>Figure 16 Composition Relationship Implementation Code</i>	50
<i>Figure 17 Class Diagram Inheritance Relationship</i>	52
<i>Figure 18 Inheritance Relationship Implementation Code</i>	52
<i>Figure 19 Class Diagram Realization Relationship</i>	55
<i>Figure 20 Realization Implementation Code</i>	55
<i>Figure 21 Association Class in Class Diagram</i>	57
<i>Figure 22 Association Class Implementation Code</i>	58
<i>Figure 23 Structural Stability Computation Steps</i>	63
<i>Figure 24 Classifier Type Changes</i>	64
<i>Figure 25 Classifier Relationships Changes</i>	64
<i>Figure 26 Class Diagram Sample version i</i>	66
<i>Figure 27 Class Diagram Sample version i+1</i>	66
<i>Figure 28 Sample Use Case with Extension Point</i>	70
<i>Figure 29 Sample Actor-Actor Relationship</i>	72
<i>Figure 30 Sample Actor Use Case Relationship</i>	73
<i>Figure 31 Use Case Generalization</i>	75
<i>Figure 32 Use Case Inclusion Sample</i>	77
<i>Figure 33 Use Case Extend Relationship Sample</i>	79
<i>Figure 34 Functional Stability Computation Steps</i>	85
<i>Figure 35 Use case Type Changes</i>	85
<i>Figure 36 Use Case Relationship Changes</i>	86

<i>Figure 37 Actor Relationships Changes</i>	86
<i>Figure 38 Use Case Sample version i</i>	88
<i>Figure 39 Use Case Sample version i+1</i>	88
<i>Figure 40 General Description of Participant</i>	92
<i>Figure 41 Sequence Diagram Participants</i>	93
<i>Figure 42 General Description of Message</i>	95
<i>Figure 43 Message Arguments</i>	95
<i>Figure 44 Synchronous Message</i>	96
<i>Figure 45 Implementation Code of Synchronous Message</i>	97
<i>Figure 46 An Asynchronous Message</i>	97
<i>Figure 47 Implementation Code of an Asynchronous Message</i>	98
<i>Figure 48 Return Message</i>	99
<i>Figure 49 Creation Message</i>	100
<i>Figure 50 Destruction Message</i>	100
<i>Figure 51 Implementation Code of Creation Message</i>	101
<i>Figure 52 Behavioral Stability Computation Steps</i>	108
<i>Figure 53 Message Receiver Changes</i>	108
<i>Figure 54 Message Caller Changes</i>	109
<i>Figure 55 Message Type Changes</i>	109
<i>Figure 56 Message Order Changes</i>	109
<i>Figure 57 Sequence Diagram Sample version i</i>	111
<i>Figure 58 Sequence Diagram Sample version i+1</i>	111
<i>Figure 59 ATM Class Diagram version 1</i>	125
<i>Figure 60 ATM Class Diagram v</i>	126
<i>Figure 61 ATMStartUp Sequence Diagram version 2</i>	131
<i>Figure 62 ATMStartUp Sequence Diagram version 2</i>	132
<i>Figure 63 Deposit Sequence Diagram version 1</i>	135
<i>Figure 64 Deposit Sequence Diagram version 2</i>	135
<i>Figure 65 Withdrawal Sequence Diagram version 1</i>	138
<i>Figure 66 Withdrawal Sequence Diagram version 2</i>	138
<i>Figure 67 SCM Class Diagram version 1</i>	142
<i>Figure 68 SCM Class Diagram version 2</i>	142
<i>Figure 69 Purchase Sequence Diagram version 1</i>	145
<i>Figure 70 Purchase Sequence Diagram version 2</i>	145
<i>Figure 71 Replenish Sequence Diagram version 1</i>	147
<i>Figure 72 Replenish Sequence Diagram version 2</i>	147
<i>Figure 73 Source Sequence Diagram version 1</i>	149
<i>Figure 74 Source Sequence Diagram version 2</i>	149

Figure 75 ORA Use Case Diagram version 1 152
Figure 76 ORA Use Case Diagram version 2 153
Figure 77 O-RED Class Diagram version 1..... 159
Figure 78 O-RED Class Diagram version 2..... 160
Figure 79 HOSS Use Case Diagram version 1 164
Figure 80 HOSS Use Case Diagram version 2 165
Figure 81 ESAP Use Case Diagram version 1 171
Figure 82 ESAP Use Case Diagram version 2 172

LIST OF ABBREVIATIONS

C	:	Classifier
CT	:	Classifier Type
NC	:	Number of Classifiers
Ch	:	Change
UCC	:	Unchanged in Classifier
NUP	:	Number of Unique Properties
CR	:	Classifier Relationships
NUCR	:	Number of Unique Classifier Relationships
SS	:	Structural Stability
U	:	Use Case
A	:	Actor
NUUP	:	Number of Unique Use Case Properties
NUAR	:	Number of Unique Actor Relationship
UN	:	Use Case Name
AN	:	Actor Name
NU	:	Number of Use Cases

NA	:	Number of Actors
UCU	:	Unchanged in Use case
UCA	:	Unchanged in Actor
FS	:	Functional Stability
UT	:	Use Case Type
UR	:	Use Case Relationship
P	:	Participant
MN	:	Message Name
MR	:	Message Receiver
MC	:	Message Caller
MT	:	Message Type
MO	:	Message Order
NM	:	Number of Messages
UCM	:	Unchanged in Messages
NMP	:	Number of Message Properties
BS	:	Behavioral Stability

ABSTRACT

Full Name : Amjad Abu Hassan
Thesis Title : TOWARDS THE DEFINITION OF SOFTWARE MODEL
STABILITY METRICS
Major Field : Software Engineering
Date of Degree : May 2015

Software metrics have become an essential part of software development due to their importance in reducing cost, effort, and time during the development phase. Many metrics have been proposed to assess different software quality attributes; stability is one of these attributes. A number of software stability metrics have been proposed at class, architecture and system levels. However, mostly, these metrics have targeted the source code.

The objective of this research is to propose software stability metrics at a model level for the UML class diagram, UML use case diagram, and UML sequence diagram. These three diagrams represent the most common diagrams in the three UML views: the structural, the functional, and the behavioral. In this research, we introduced a new assessment approach called the Client Master Approach to skip duplication. The assessment methodology we followed for tracking changes is: analysis of each UML diagram, applying the client master approach, and getting the change possibilities. Based on the assessment process, a new suite of metric was proposed; a metric for the UML class diagram, a metric for UML use case diagram, and a metric for the UML sequence diagram. Validation of the proposed metrics suite was performed, theoretically and empirically. Theoretically, using the metric-evaluation framework. We apply our metrics on six different case studies that represent multi UML diagrams.

ملخص الرسالة

الاسم الكامل	:	امجد ابو حسان
عنوان الرسالة	:	تعريف مقاييس لثباتية نماذج البرمجيات
التخصص	:	هندسة برمجيات
تاريخ الدرجة العلمية	:	مايو 2015

مقاييس البرمجيات اصبحت جزء مهم في عملية تطوير البرمجيات نظرا لاهميتها في تقليل التكلفة والجهد والوقت اللازم لعملية التطوير. العديد من المقاييس تم استحداثها لقياس مدى كفاءة البرامج . الثباتية هي واحدة من خصائص البرمجيات التي يمكن قياسها وتقييمها. تم انشاء العديد من المقاييس لقياس ثباتية النظام والهيكلية والصفوف. لكن معظمها كان موجها لقياس ذلك على مستوى الكود.

الهدف من هذه البحث هو انشاء مجموعة مقاييس لقياس الثباتية على مستوى نماذج البرمجيات المسمى UML وهذه النماذج هي Class و Sequence و Use Case . هذه النماذج هي المستخدمة غالبا لتمثيل اي برنامج وهي تمثل ثلاثة اتجاهات مختلفة وهي اتجاه الهيكلية واتجاه الوظيفية واتجاه السلوك.

في هذه البحث تم تقديم منهجية جديدة لتقييم النماذج , سميت منهجية الخادم والسيد. هدفها التخلص من احتساب التغيرات اكثر من مرة خاصة في العناصر التي لها كثير من العلاقات. اجراء عملية التقييم الذي اتبعناه يبدأ بتحليل النماذج ثم تطبيق منهجة الخادم والسيد , وبعد ذلك نحصل على كل التغيرات الممكنة في النموذج. في النهاية تم استحداث مقاييس جديدة لكل من Class و Sequence و Use Case وتم التحقق من هذه المقاييس نظريا وعمليا.

CHAPTER 1

INTRODUCTION

Software metrics are units of measurements that are useful for measuring quality, performance, debugging, management, and estimating costs [1]. The collected measurements give an overview about the software project, and show a clear image about the current situations, which help in making quantitative/qualitative decisions during the software lifecycle.

Software systems are becoming more and more sophisticated. Writing newer versions has become complex due to stakeholders' changing demands, thus the maintainability is essential as it is a costly process. ISO 9126 characterizes maintainability with four sub-characteristics, one of which is stability.

Mitigating the evolved changes is very important for software developers in order to stabilize a system and preserve its design. Therefore, the need of stability measurements is very important. Many software metrics have been proposed to cover this area. Most of these metrics have been introduced to assess the stability at the code level. However, little research has been done to measure stability at the models level.

In this research, we propose a metrics suite that addresses model stability. We will cover three UML diagrams. These three diagrams represent the three main views of UML diagrams, which are: the class diagram that represents the structural view, the sequence diagram that represents the behavioral view, and the use case diagram that represents the functional view [2].

1.1 Motivation

Stability is an important quality attribute in software development, which gives an overall assessment of the software system, ISO 9126 defines many quality characteristics and sub-characteristics that need a huge number of metrics to cover these attributes. ISO 9126 has six main characteristics: functionality, reliability, usability, efficiency, maintainability, and portability. Maintainability consists of four sub-characteristics, one of which is stability.

Maintenance emerged from the volatility of requirements, and the increasing change demands from customers and stockholders; this affects the software system. Development has to keep up with requirement changes, as well as other implementation issues like technologies and different platforms; the software should be designed to accommodate these changes.

Stability, is defined by Daskalantonakis [3] as “a method of quantitatively determining the extent to which a software process, product, or project possesses a certain attribute”. Azuma et al. [4], defines the metric as "a quantitative scale and method which can be used to determine the value a feature takes for a specific software product". In our assessment, we will measure the amount of unchanged in the UML diagrams.

Stability plays a main role in indicating and evaluating the maintenance process, its effort and cost. Unstable software may lead to high cost and effort of maintenance, user dissatisfaction, poor deliverables quality and other issues.

Measuring stability, especially at the design level, provides an early estimation of the project and thus an early judgment about the software status and the next movements toward enhancing performance, the development process, and mitigating the changes.

The literature shows that all studies focus on assessing code stability; therefore, due to the lack of stability metrics at the model level, we plan to propose a suite of stability metrics that covers different UML views.

1.2 Research Objectives

The objective of this research is to *propose a metrics suite that measures the stability of different UML models*. We will select one diagram from each UML view, namely:

- **Class Diagram:** this diagram describes the structure of a system, by showing the system's classes, their attributes, variables, methods, and the relationships among objects.
- **Sequence Diagram:** a sequence diagram is the interaction diagram that represents the sequence of messages exchanging between objects to implement a specific scenario.
- **Use Case Diagram:** a use case diagram is used to represent system functionality. Use cases describe the interaction between customers and the system, by providing a graphical representation of what the system exactly does.

1.3 Contributions

The deliverables and contributions of this research are:

- **Stability Metrics Survey:** this classifies a wide range of existing software stability metrics, which focus on object-oriented diagrams metrics (class, sequence, and use case). The survey will cover all the characteristics and properties for those metrics.

- **Similarity Metrics Survey:** this covers a wide range of existing software similarity metrics, which focus on object-oriented diagram metrics (class, sequence, and use case). The survey will illustrate all techniques and approaches that have been used in similarity assessment.
- **Structural Stability Metric:** this proposes a metric to assess the UML class diagram's stability that covers all structural properties.
- **Functional Stability Metric:** this proposes a metric to assess the UML use case diagram's stability that covers all functional properties.
- **Behavioral Stability Metric:** this proposes a metric to assess the UML sequence diagram's stability that covers all behavioral properties.

1.4 The Outline

The rest of this thesis is structured as follows: Chapter 2 presents a the background for the UML and its main views, and explains the selected diagrams, the UML class diagram, the UML use case diagram, and the UML sequence diagram. Chapter 3 surveys the proposed stability metrics and the similarity metrics. Chapter 4 presents the research methodology through assessment. Chapter 5 introduces the structural metric that is proposed for the UML class diagram. Chapter 6 discusses the functionality metric that is proposed for the UML use case diagram. Chapter 7 introduces the behavioral stability metrics for the sequence diagram. Chapter 8 contains the theoretical and validation of the structural stability metric, the functional stability metric and the behavioral stability metric. Chapter 9 presents the case studies. Chapter 10 discusses the conclusion and the future work.

CHAPTER 2

BACKGROUND

This chapter introduces a background of some of the concepts used in this research. The background highlights the different UML models that have been selected.

2.1 Unified Modeling Language (UML)

Unified Modeling language (UML) is a standard notation for the modeling language. UML enables developers to visualize software systems artifacts. The objective of UML is to provide a standard way to visualize the system design. It was appeared in the early nineties by Grady Booch, Ivar Jacobson and James Rumbaugh, and later in 1997 the Object Management Group (OMG) adopted UML as Object-Oriented design and analysis language.

Since that OMG developed and enhanced the UML, many versions have been released; the last one is UML 2.5, which is still under construction. Nevertheless, in our research we will use the stable and most used version, the UML 2, which has become the standard industry modeling language.

The Object Management Group (OMG) defines UML as:

“The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system.” [5]

UML is used to represent the three main views of the systems: structural, functional and behavioral. Each view can be represented by different UML diagrams, as in Figure 1.

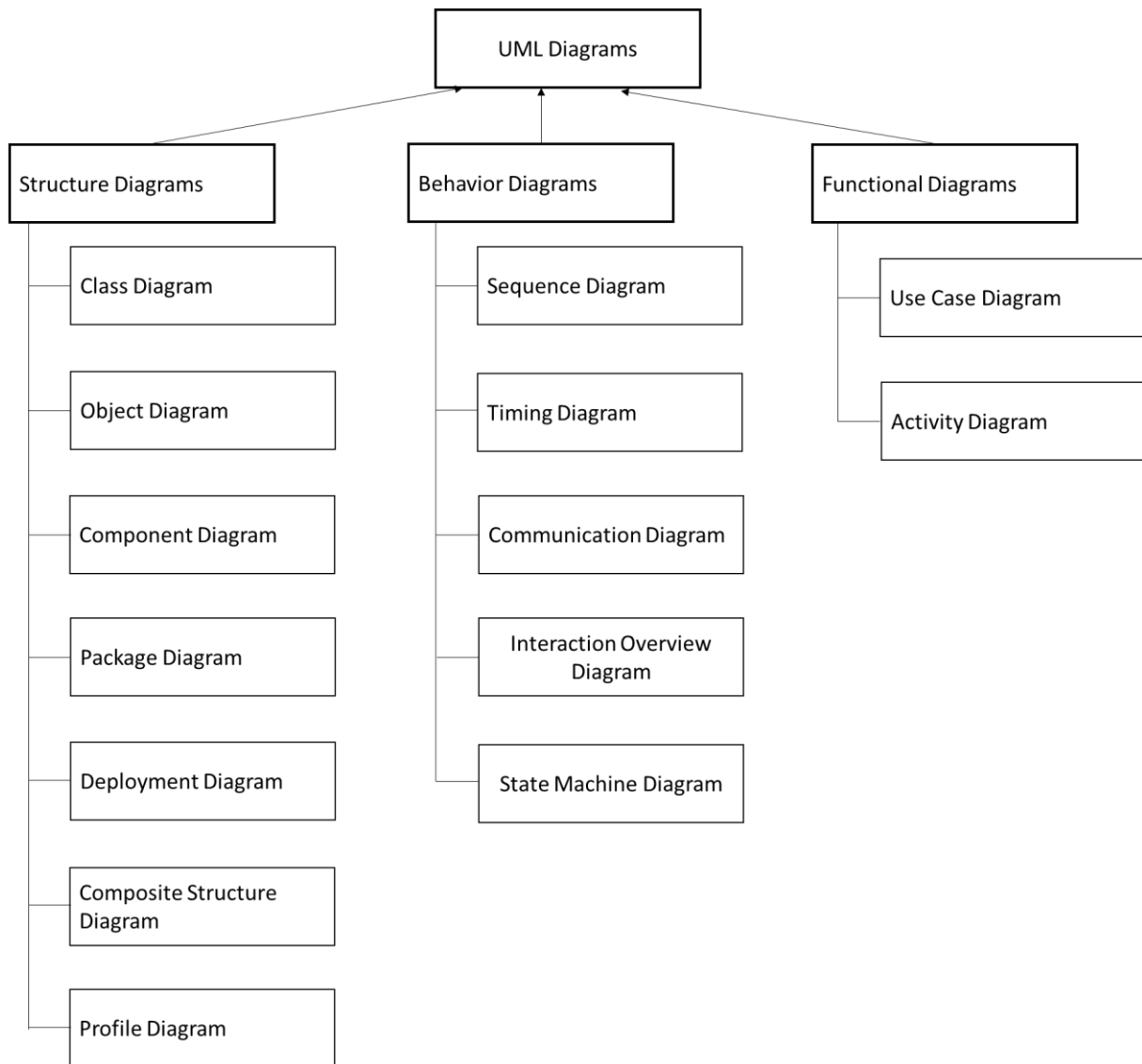


Figure 1 Hierarchical Classification of UML Diagrams [6]

2.1.1 Structural View

Structure diagrams describe the static aspects of the system. They are used extensively in documenting the software architecture of software systems. Objects and classes are the basic

building elements in an object-oriented design. These elements represent the system concepts, which include abstract and implementation concepts. The structural view has different diagrams used to capture the physical organization of the system elements.

UML has the following seven types of structural diagrams. A class diagram is the most commonly used one. The list of all UML structure diagrams is:

- Class Diagram.
- Package Diagram.
- Deployment Diagram.
- Component Diagram.
- Composite Structure Diagram.
- Object Diagram.
- Profile Diagram.

2.1.2 Behavioral View

Behavioral diagrams describe the behavior, dynamic features and methods of the modelled structural objects of the systems. A sequence diagram is the almost commonly used diagram to model the behavior of the system. UML models applicable to this view include:

- Sequence Diagram.
- Timing Diagram.
- State Machine Diagram.
- Communication Diagram.

- Interaction Overview Diagram.

2.1.3 Functional View

Functional diagrams describe the systems from a user's perspective. They show how the system is supposed to work by describing the system functionality from the user's perspective. A use case diagram is the commonly used diagram to model system functionality. It is one of the diagrams we selected from the functional view to measure its stability. The list of all UML functional diagrams is:

- Use Case Diagram.
- Activity Diagram.

2.2 UML Class Diagram

Class diagrams are the most popular and most common UML diagrams. They are used to capture the static relationships of the object-oriented systems and represent its structural view. A class diagram comprises a set of classes that represent the core of any object-oriented system. It also consists of different types of relationships used to connect classes together.

A class diagram consists of two main parts: the classes, and the relationships between these classes. Each part has its own properties and types. Classes are identified by name, and have an access level, a set of variables and methods. Variables have a name, access level and data type. Methods have a name, access level, return type, and parameters which also have their own properties.

The second part, is the relationships, which have different types: dependency, aggregation, composition, inheritance, realization, and association.

2.2.1 Classes

Class diagram classes consist of a set of objects that share attributes and methods. Classes are represented by a rectangle that has three parts. The first part contains the class name. The second part contains the attributes, their names, visibility, and data types. The third part contains the operations, their names, signature, visibility, and return type.

2.2.2 Class Diagram Relationships

Relationships allow classes to interact: there are different types of relationships with different purposes and strengths. By strength we mean the level of dependency on of two classes involved in this relationship. There are six main relationships, shown in Figure 2:

- **Dependency Relationship**

Dependency between two classes which represented by a dotted line arrow, declares that one class (target class) depends upon another class (source class). The relationship means that the target class needs information from the source class.

- **Association Relationship**

An association between two classes, represented by a solid line, declares that objects of each class depend upon the objects of the other class. Association means that a class will actually contain a reference to an object, or objects, of the other class in the form of an attribute.

- **Aggregation Relationship**

Aggregation between two classes, represented by an empty diamond, declares that one class (whole- a class with the diamond edge) is an aggregate of the other class (part) objects. Aggregation is a stronger version of association, thus it is a one-way association.

- **Composition Relationship**

Composition between two classes, represented by a filled diamond, declares one class (whole-a class with the diamond edge) is composing the other class (part) objects. Composition is a stronger version of aggregation. In this relationship, the part class lifetime depends on the whole class lifetime.

- **Inheritance Relationship**

Inheritance or generalization is a relationship between a class (super) and a subclass. In this relation the subclass inherits the parent class structure. Inheritance between two classes, represented by an empty triangle arrowhead, means that one class is a type of another one.

- **Realization Relationship**

Realization between two classes, represented by a black triangle arrowhead, means that one class realizes another one.

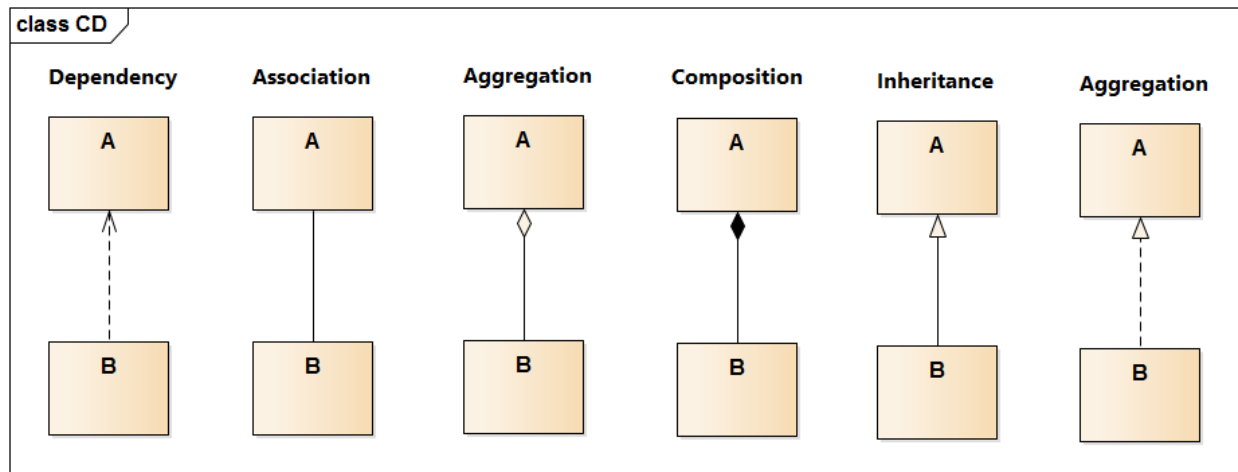


Figure 2 Class Diagram Relationships

2.2.3 Association classes

These classes are new classes; they can be introduced by the association itself. Association classes are particularly useful in complex cases. They are used when a class is linked to two classes because those two classes have a relationship with each other.

2.3 UML Use Case Diagram

Use case diagrams represent a system's functionality; they are used to model a functional requirement. Use case was introduced by Jacobson [7] and later added to the UML group by OMG. They describe the system's requirements from the user's point of view, by identifying the system deliverables to the users [8]. Use case diagrams mainly consist of the use cases that represent the functionality, and the actors who invoke these functionalities.

2.3.1 Use Cases

The use cases are used to describe a specific object-oriented system functionality. The use case name itself is used as a description of functionality. There are two ways for use case representations in UML. One way is by using an oval; the other way is by using a classifier notation.

2.3.2 Actors

The actor is the one who initiates the use case. Actors have different ways of being drawn. One way uses a stick man figure. Also, classifier notation can be used to represent the actor. The actor can have relationships with another actor or with use cases.

2.3.3 System Boundaries

A system's boundaries are used to contain all system functionality (use cases). Anything else should be modeled out of the system as an actor. The boundaries are represented by a simple rectangle.

2.3.4 Use Case Diagram Relationships

Relationships allow use cases to interact. There are three main relationships, as shown in Figure 3:

- **Use Case Generalization**

Use case generalization is like inheritance in class diagrams. It is typically used to describe high level functionality, without going into details.

- **Use Case Inclusion**

Use case inclusion is used to share functionality by grouping several use cases to include a common one. However, this general use case is not complete on its own.

- **Use Case Extension**

A use case extension is used in the case of inserting, a further functionality to the base use case. This is done if conditions are met. In this case the original use case has to be complete on its own. Usually the extending use case has a smaller scope.

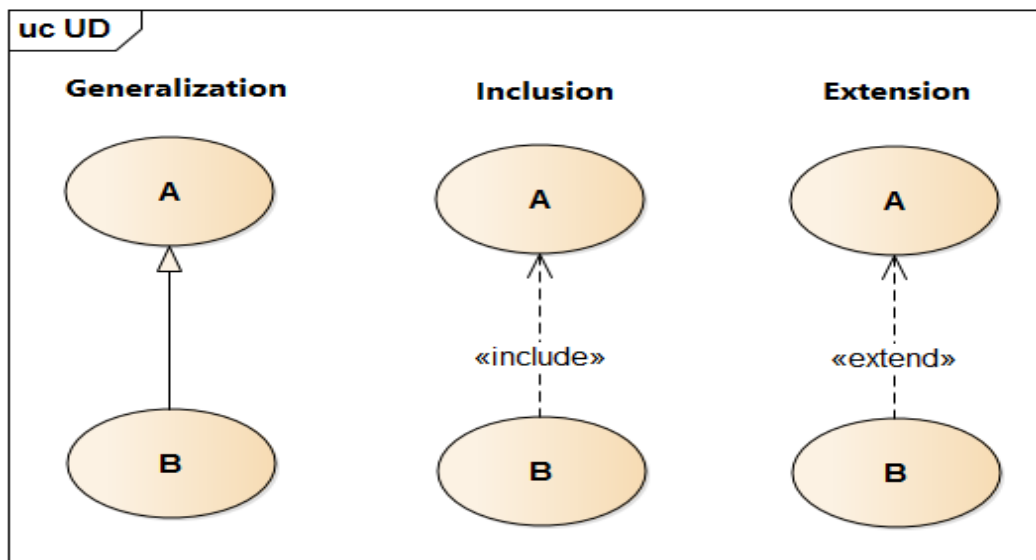


Figure 3 Use Case Diagram Relationships

2.4 UML Sequence Diagram

The sequence diagrams are a graphical representation of the control flow. They are particularly useful for describing executions that involve several classes. A sequence diagram is used to capture order of interactions between different system parts, and describes which interaction will occur if a particular event is triggered. It also shows different information about the system interactions.

A sequence diagram is made up of a collection of participants, lifelines, and messages.

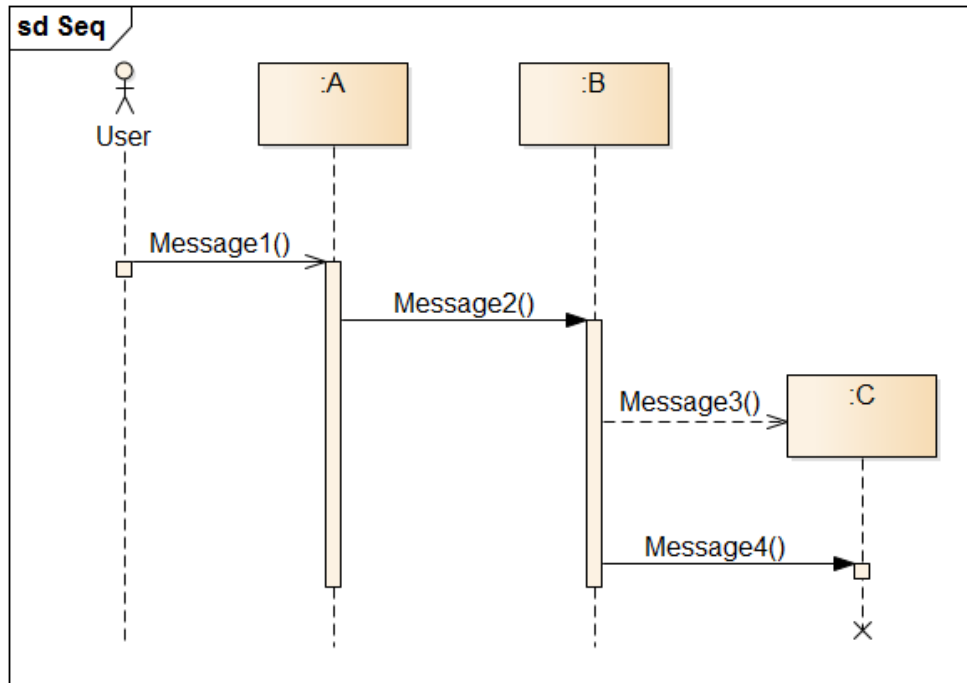


Figure 4 Sequence Diagram Sample

2.4.1 Participant

Each participant has a corresponding lifeline, a solid vertical line. The lifeline indicates the classifier location in the sequence. From Figure 4, A represents a sequence participant.

2.4.2 Time

What we need from the time here is the sequence diagram's interactions order. Time starts at the top of the sequence diagram and then progresses down in the sequence diagram.

2.4.3 Messages

Messages are the sequence diagram's building blocks. They represent the interaction points. The interaction happens when a participant sends a message to another participant. Messages are

expressed as an arrow from the Message Caller to the Message Receiver. They have no specific direction; they can be right to left, left to right, or from and to the same Message Caller.

2.4.4 Notes

Notes are used to describe the diagrams, and hold some information about them, like local variables' names, the values, and can state invariant information.

2.4.5 Messages Type

A sequence diagram has five types, and each type has its own meaning, shown in Figure 4:

- **Synchronous messages**

A synchronous message is used in a waiting case, when the Message Caller waits for the return values after the invocation of the Message Receiver. This can be implemented in the code as a simple method invocation.

- **Asynchronous messages**

In this type, when the message is invoked, the Message Caller does not wait for the message invocation to return; it moves on with the rest of the interaction's steps. This means that the Message Caller will invoke a message on the Message Receiver and the Message Caller will be busy invoking further messages before the original message returns. It can be named as a "fire and forget" message.

- **Return messages**

The return message is an optional piece of notation that can be used at the end of an activation bar to show that the control flow of the activation returns to the participant that passed the original message.

- **A participant creation message & a participant destruction message**

Participants do not necessarily live for the entire duration of a sequence diagram's interaction. Participants can be created and destroyed according to the messages that are being passed.

CHAPTER 3

LITERATURE REVIEW

This section introduces a survey on software stability and similarity, at the design and code levels. The literature highlights the existing proposed metrics used to evaluate software stability, and the techniques used to assess software similarity.

3.1 Stability

This section presents a survey of existing stability metrics that are distributed on three levels: architecture, class, and system.

3.1.1 Architecture Level Metrics

Sethi et al. [9] devised a metrics suite to measure software modularity and stability at the architecture level; the new metrics suite takes into consideration the environmental conditions.

Sethi et al. proposed the Decision Volatility metric to assess decisions that may be affected by the environmental conditions (Envr Impact). The metric's value indicates the amount of change on the software; more changes lead to more impact on the stability.

Molesini et al. [10] analyzed aspect-oriented composition mechanism's influence on a modules architectural stability. The authors investigated to what extent aspect-oriented architectures are stable when the change occurs, and they found that Aspect-Oriented (AO) architecture is more

stable when a change targeted a crosscutting concern. They used a conventional set of metrics to quantify change propagation in AO architecture. These metrics depend on collecting a number of components that had been added or changed, the connectors that had been added or changed, and the number of point cuts that had been added or changed.

Tonu et al. [11] introduced the architectural stability approach. This approach makes use of metrics and combines retrospective and predictive evaluation. The retrospective approach evaluates architectural perspectives of stability by analyzing the successive releases of a software system, while predictive evaluation checks the potential changes. The metric-based approach has been used to make a late evaluation by extracting the architecture from the source code first, and then applying retrospective and predictive analyses.

Jazayeri [12] evaluated structural stability using retrospective analysis. It is done by applying three kinds of retrospective analysis: 1) analysis using basic measurements like the number of modules changed, module size ...etc., 2) by indicating the coupling among system modules, and 3) one by mapping out system evolution using color visualization.

Bansiya [13] calculated the extent of change between two software versions. He presented a methodology to assess framework architecture stability by using an Object-Oriented (OO) metrics suite that evaluates framework structural characteristics. These characteristics are: design size (in number of classes), number of class hierarchies, number of multiple inheritances, number of single inheritances, average depth of class inheritance hierarchies, average width of class inheritance hierarchies, number of parents, number of methods, and class coupling. After computing these characteristics' metric values, the extent-of-change is identified by normalizing these values with respect to earlier versions' values, and calculating the difference between aggregate-change values,

between subsequent releases and the version i release. The aggregate-change is the sum of all characteristics' values in the same version. The extent-of-change values indicate that the higher the value the more unstable the system is.

Haohai et al. [14] used Bansiya's [13] approach. In addition they proposed another six metrics and followed the same evaluation procedure: These metrics are: the average number of additional operations, average number of stereotypes, number of abstract meta-classes, average number of well-formed rules, number of concrete meta-classes, and the number of meta-classes which have no parent and no child in the meta-model.

Mattsson and J. Bosch [15] also used the same methodology presented by Bansiya [13], but they applied it on a different suite of metrics.

Moataz et al. [16] introduced a way for measuring architectural stability by defining a release's similarity to the base version. They proposed two similarity metrics, Shallow Semantic Similarity Metric (SSSM) and Relationship-Based Similarity Metric (RBSM); hence, greater similarity leads towards better stability. SSSM computes the average similarity between two pairs of classes by comparing successive releases' architecture with the base version architecture. The RBSM similarity measurements are based on comparing the existing inheritance relationships among two models classes.

Hassan [17] proposed a metrics suite to measure architecture stability, which are the inter-package and intra-package set of metrics. The inter-package set of metrics considers the connections between elements of two different packages. Intra-package metrics consider the connections between elements in the same package. Metric calculation depends on an element's change indication. These change possibilities are: modification, no change, addition, and deletion.

Aversano et al. [18] defined two metrics, CDI, and CCI, to assess the architecture stability. Core Design Instability (CDI) is used to indicate the changes that affect the core architecture. It is computed as follows:

$CDI = (b+c)/m$, where,

m: number of packages that belong to the extended core of release N,

b: number of new packages that are added to the extended core,

c: sum of packages that belong to extended core N, and which do not belong to extended core N+1.

Core Calls Instability (CCI) was proposed to evaluate the package's interactions change. It is calculated as follows:

$CCI = (x+y)/z$, where,

z: the total number of calls between packages that belong to the extended core of release N,

x: the total number of new calls between packages that belong to the extended core of release N+1,

y: the total number of calls between packages of the extended core of release N and which are not present in the extended core of the release N+1 after the executed changes.

3.1.2 Class Level Metrics

Grosser et al. [19] proposed a metric to assess class stability based on case-based reasoning (CBR). The authors used CBR to identify quality challenges, and evaluate them using several metrics related to four categories, which are complexity, inheritance, cohesion, and coupling. The

evaluation results are then compared to other nearest known software items in order to predict the stability.

Grosser et al. [20] included another factor called stress, which results from a primary change in the requirements. The stress factor is computed at the class level between two software versions.

Rapu et al. [21] presented an approach that depends on historical information to detect class problems, such as God Classes, and Data Classes. Rapu et al. indicated that the class is stable if there is no difference in measurements between version $i - 1$ and version i . The authors did not take into account the class changing size; they considered the class to be changed if a method was added or removed.

Li et al. [22] introduced three metrics, which are: Class Implementation Instability (CII), System Design Instability (SDI), and System Implementation Instability (SII) to assess Object-Oriented (OO) stability at the implementation level. CII was introduced to measure changes from design N to design $N + 1$ during object-oriented implementation at the class level. The preceding is computed by calculating the percentage of LOC changes between the two versions.

Alshayeb et al. [23] proposed a Class Stability Metric (CSM) to assess stability at the class level. The authors selected eight different class properties to evaluate stability. These properties are: the class access-level, the class interface name, the method access-level, the inherited class name, the method signature, the class variable, the class variable access-level, and the method body.

CSM follows property change (addition, deletion, modification, and unchanged) between the two versions $i + 1$ and i , if there is no change then the class is stable. Later on, Alshayeb [24] introduced a minor modification to the CSM by considering the changes between the $n+1$ and n versions, instead of the base version.

Elish and Rine [25] investigated process-related and product-related indicators that affect structural stability measures. Elish and Rine selected several metrics suites to gather data about version i of the software and used these data to predict structural stability in version $i+1$. Elish and Rine measured stability from two perspectives. The first perspective considered how much of the base design structure remained unchanged, while the second perspective considered how long the structure remained invariant. Sixteen metrics were proposed to define the number of classes that were modified, added, deleted, and unchanged. In addition, they are used to specify the relationship types of the classes, which can be generalization, aggregation, dependency or association.

Mattsson and Bosch [26] introduced a relative-extent-of-change metric and used Bansiya's [13] stability assessment method in order to evaluate software systems. They used different sets of metrics suites to evaluate structural, functional, and relational characteristics.

Elish and Rine [27] investigated the relationship between the C&K metrics [28] and the logical stability. Their investigation found a good correlation between CBO and RFC metrics with logical stability. They also found a negative correlation of WMC, DIT, CBO, RFC, and LCOM metrics with logical stability, and no correlation in the NOC case. Elish and Rine used an algorithm to compute the program's logical stability. The algorithm applies all potential class level changes to the other design classes and calculates the ratio of the number of times the class is impacted by the total number of possible changes. Class level changes are: Data type, Delete, Scope (protected to private), Scope (public to private), Scope (public to protected), and Return data type. For class methods: Delete, Scope (protected to private), Scope (public to protected), and Scope (public to private).

3.1.3 System Level Metrics

System Implementation Instability (SII), mentioned earlier, was proposed by Li et al. [22] to measure changes from design N to design N + 1 during object-oriented system implementation. SII is computed by calculating the percentage of LOC changes between two versions in the entire system.

Raemaekers et al. [29] proposed four metrics to evaluate implementation and public interface in order to indicate library stability. These four metrics are: Weighted Number of Removed Methods (WRM), which is used as a measure for interface stability; the Amount of Change in Existing Methods (CEM), which indicates the amount of change in existing methods; Ratio of Change in New to Old Methods (RCNO), which indicates the amount achieved of work, and Percentage of New Methods (PNM), which computes the percentage of the new added methods.

Kelly [30] investigated software systems in order to indicate the systems that have been maintained actively. Kelly proposed a method for inspecting such systems by using stability as an indicator of the design characteristics that affect the maintainability. The author used different metrics to assess design characteristics, and to find the difference between two software versions. These metrics are: total number of common blocks (CB), total lines of code (LOC), total number of common block variables (VAR), and total number of modules (MOD).

Yau and Collofello [31] measured program and module logical stability. The logical stability is a measure of the change impact of a module to the other modules in the program. The authors calculated the logical ripple effect of a primitive modification to a program.

Their formula depends on computing the modification probability that equals one divided by the number of variable definitions in the module, and the sum of McCabe's Cyclomatic number.

Li et al. [22] proposed the System Design Instability (SDI), specified for assessing Object-Oriented (OO) at the implementation level. SDI is used to capture changes of software design by measuring the percentage of change from design N to design N+1. SDI considers: change percentage of newly added classes, classes with changed names, and deleted classes.

SDI is computed as follows:

$SDI = [(a + b + c)/m] \times 100$, where:

a: change in classes' name,

b: added classes,

c: deleted classes,

m: number of classes in design N.

The SDI value is greater than or equal to zero, where zero means that the design is stable.

Later, Alshayeb and Li [32] redefined SDI considering a fourth aspect of changes, which is the percentage of change in inheritance hierarchy. The new formula is computed as follows:

$SDI = [(a + b + c + d)/m] \times 100$, where,

d represents change in inheritance hierarchy.

Olague et al. [33] introduced the SDI_e metric by recasting the System Design Instability (SDI) proposed by Li et al. SDI_e is based on maximum system entropy, and considers some different aspects of input which are: the added classes, the deleted classes, changed classes, and the unchanged classes. The Entropy-based SDI metric (SDI_e) is computed as follows:

$$SDI_e = - \sum_{i=1}^j \frac{C_i}{N} \log_2 \frac{C_i}{N}$$

j: the total number of categories of SDI_e, which has four categories: added, deleted, changed, unchanged.

C_i: the classes' count in category i, and N represents the total number of system classes.

Martin and Martin [34] proposed a metric to evaluate the components' stability based on the total number of dependencies that enter or leave the component. It is computed as follows:

Instability = (C_e) / (C_a + C_e), where,

C_a: the total number of classes in other components that depend upon classes within the component,

C_e: the total number of classes in other components that the classes in the component depend upon.

The metric values range from 0 to 1; 0 indicates that the component is stable. Thus the system will be stable if the maximum number of components is stable.

Table 1 provides an overview of the surveyed metrics. The first column shows the reference. The second column lists the metrics assessment level (class, system, or architecture). The third column presents the artifact used to compute the metrics. The fourth column shows the number of properties used in calculation. The fifth column is the validation techniques. The last column shows a brief description of the metric.

3.2 Similarity

Mayrand et al. [35] used several metrics to compare functions in order to identify duplication and cloning level, based on computing four points: name, layout, expressions, and control flow.

Patenaude et al. [36] extended the Bell Canada Datrix tool [35] to find Java clones. The authors used several complexity metrics to evaluate methods; methods that have similar metrics values are clones.

Kontogiannis et al. [37] introduced two techniques for clone detection. In the first technique, they selected five well-known metric suites that capture code information and applied them on the two code fragments to compare their values.

In the second technique, they used dynamic programming (DP) to compare two code segments to compute what is called distance, based on insertion, deletion, and operation comparison.

Balazinska et al. [38] applied a similar method in their similar methods classifier (SMC) tool. The authors represented the code in abstract syntax tree (AST) and performed code segmentation, and then they applied dynamic programming (DP).

Qiu et al. [39] introduced a metric to assess software similarity by quantifying the nodes and edges of the class diagrams. The authors calculated software similarity by computing structural similarity and property similarity. First, they constructed class diagrams from the source code. The class is represented by node, and the relationship is represented by edge, where the authors assign weight to edges based on coupling metrics. Finally, the similarity between the nodes and edges is computed using the iterative method.

Krinke [40] extracted a program dependency graph from the source code, and detected the similarities between the subgraphs using the iterative approach. The nodes represent the expressions and the statements, while data dependencies are represented by edges.

Liu et al. [41] proposed a plagiarism detector based on a program dependency graph (PDG).

Johnson [42] used fingerprinting to find matches in source code text. Fingerprinting methodology converts a substring of the code to hash, where each two code segment's hash values are matched.

Li et al. [43] introduced the CP-Miner tool, which is a token-based tool used to find copy and paste in source code. The token-based tool finds the similar sequences that appear in the same order in the code using repeated subsequence data mining.

This literature presents many metrics that have been used to compute software similarity and code clones. The surveyed literatures declared five main techniques to measure software similarity.

These techniques were distinguished using the analysis methodology. The five techniques are:

Text-based approach, which depends on natural language processing to find a repeated fingerprint to use in code segments matching.

Token-based approach, which converts the source code into a sequence of tokens, and then scans them for repeated subsequences.

Tree-based approach, which transforms the source code program into an abstract syntax tree (AST) and uses tree-matching to find similar sub-trees.

Graph-based approach, which extracts a program dependency graph from the source code and detects sub graphs' similarities using the iterative approach.

Metric-based approach, which applies a number of metrics on the code, where they are used to compare different code segments.

Table 2 provides a high-level overview of the surveyed techniques, tools, and metrics. The first column shows the citation(s), while the second column shows the used approach and the last column shows a brief description of the citation.

3.3 Summary

Table 1 summarizes the investigated stability metrics and reveals that no metrics exist to measure the software model's stability. Metrics are used to evaluate code stability. Assessment covers three levels, which are the architecture, the class, and the system.

Most of the surveyed metrics were validated either empirically or theoretically; they were validated empirically using case studies or experiments. However, few of them were validated theoretically.

Table 2 shows the different techniques used to evaluate the similarity between software systems, which are: text-based approach, token-based approach, tree-based approach, graph-based approach, and metric-based approach.

This literature presents many metrics that are used to assess software stability. We noticed that the main metrics focus on assessing code stability.

The surveyed approaches from literature show that no research investigated measuring stability for individual design models. Therefore, the objective of this research is to propose a set of metrics to measure the stability of UML class, sequence, use case, and the integrated model.

Table 1 Stability Survey Summary

Reference	Metric Level	Artifact	Language Independent	Set of properties or metrics	Validation	Metric Description
Grosser et al. [19]	Class	Code	Yes	22 old metrics	Experimental	Evaluating stability using several metrics related to the four categories: coupling, cohesion, inheritance and complexity
Grosser et al. [20]	Class	Code	Yes	14 old metrics	Experimental	Evaluating stability using several metrics related to the four categories: coupling, cohesion, inheritance and complexity
Rapu et al. [21]	Class	Code	Yes	9 old metrics	Case Study	Checks if class is stable or not
Li et al. [22]	Class	Code	Yes	1 property	Theoretical & Experimental	Calculates the percentage of LOC changes between two versions
Alshayeb et al. [23], Alshayeb [24]	Class	Code	Yes	8 properties	Theoretical & Experimental	CSM metric follows the change in properties (addition, deletion, modification, and unchanged) between two versions
Elish and Rine [25]	Class	Code	Yes	17 new metrics and 16 old metrics	Case Study	Investigates product-related and process-related indicators that affect structural stability measures
Li et al. [22]	System	Code	Yes	1 property	Theoretical & Experimental	Calculates the percentage of LOC changes between two versions

Raemaekers et al. [29]	System	Code	Yes	4 properties	Experimental	Indicates library stability, by calculating stability of implementation and public interface of the library
Kelly [30]	System	Code	Yes	4 properties	Case Study	Computes stability by finding the difference between two software versions
Yau and Collofello [31]	System	Code	Yes	2 properties	Theoretical & Case Study	Computes the modification probability
Sethi et al. [9]	Architecture	Code	Yes	6 properties	Case Study	Calculates stability by taking into consideration the environmental conditions
Molesini et al. [10]	Architecture	Code	Yes	5 properties	Experimental	Quantifies change propagation in AO architecture
Tonu et al. [11]	Architecture	Code	Yes	4 old metrics	Experimental	Combines retrospective and predictive evaluation
Jazayeri [12]	Architecture	Code	Yes	3 properties	Case Study	Evaluates structural stability using retrospective analysis
Mattsson and Bosch [26]	Class	UML	Yes	20 old metrics	Case Study	Introduces relative-extent-of-change
Elish and Rine [27]	Class	UML	Yes	10 properties	Experimental	Computes the logical stability
Li et al. [22], Alshayeb and Li [32], Olague et al. [33]	System	UML	Yes	1 property	Theoretical & Experimental	Capture changes of software design by measuring the percentage of changes from design N to design N+1

Martin and Martin [34]	System	UML	Yes	1 property	-	Measures the component stability based on dependencies
Bansiya [13]	Architecture	UML	Yes	9 old metrics	Case Study	Introduces a methodology to assess framework architecture stability based on extent-of-change
Ma et al. [14]	Architecture	UML	Yes	6 old metrics	Experimental	Uses Bansiya [13] methodology
Mattsson and Bosch [15]	Architecture	UML	Yes	20 old metrics	Experimental	Uses Bansiya [13] methodology
Moataz et al. [16]	Architecture	UML	Yes	2 properties	Case Study	Measures stability by defining the similarity of the releases to the base version.
Hassan [17]	Architecture	UML	Yes	20 property	Theoretical & Experimental	Measures stability using the inter-package and intra-package metrics
Aversano et al. [18]	Architecture	UML	Yes	6 properties	-	Proposes CDI and CCI metrics to assess architecture stability
The proposed Metric	Class Diagram	UML	Yes	9 properties	Theoretical & Case Study	Propose Structural Stability metric to assess UML class diagram
The proposed Metric	Sequence Diagram	UML	Yes	7 properties	Theoretical & Case Study	Propose Functional Stability metric to assess UML sequence diagram
The proposed Metric	Use Case Diagram	UML	Yes	9 properties	Theoretical & Case Study	Propose Behavioral Stability metric to assess UML use case diagram

Table 2 Similarity Techniques Summary

Reference	Techniques	Metric Extraction Level	Metric Description
Qiu et al. [39]	Tree-based	Code	Constructs class diagram, and use iterative method to compute similarities
Krinke [40]	Graph-based	Code	Uses program dependency graph to detect similarities
Mayrand et al. [35]	Metrics-based	Code	Identifies software functions clone level
Patenaude et al. [36]	Metrics-based	Code	Identifies software method clones
Kontogiannis et al. [37]	Metrics-based	Code	Detects clones using structure based metrics and dynamic programming (DP) techniques
Balazinska et al. [38]	Tree-based	Code	Uses dynamic programming (DP) on abstract syntax tree (AST)
Liu et al. [41]	Graph-based	Code	Uses program dependency graph to detect plagiarism
Johnson [42]	Text-based	Code	Fingerprints to find matches in source code text
Lu et al. [43]	Token-based	Code	Finds copy and paste in source code

CHAPTER 4

RESEARCH METHODOLOGY

This chapter presents the methodology used to analyze and assess the UML diagrams' stability.

According to the used definition of the stability, we are looking to compute the unchanged percentage of the UML diagrams. However, considering whether it is this part or that part of the UML which is changed is not that easy. On UML diagram elements, the decision process we use to select which element is changed is dependent on the relationships with other elements; therefore we need to analyze each element and detect the parts that are affected by the external ones. The measurement process of UML diagrams is done through three main steps: the analysis, the evaluation, and the proposal of a metric. Figure 5 shows the assessment methodology.

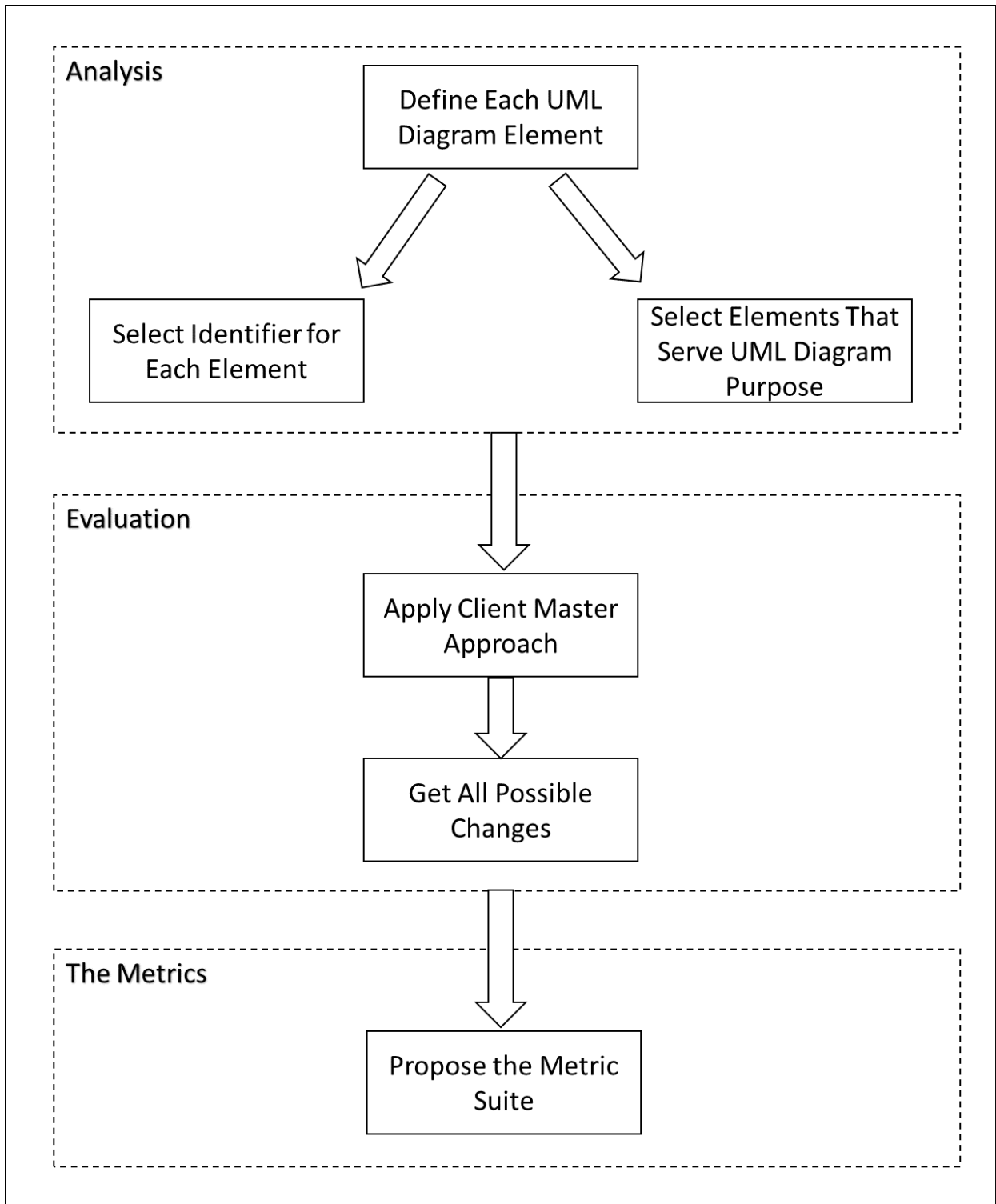


Figure 5 Research Methodology

4.1 Analysis

The analysis is the first part of the assessment methodology. The purpose of this part is to identify the three UML diagrams, the UML class diagram, the UML use case diagram, and the UML sequence diagram. Then we select each UML diagram elements for the next steps, and, in addition, we select an identifier for each one.

First, we collect all the available information about each UML diagram by identifying all the elements of the UML class diagram, the UML use case diagram, and the UML sequence diagram, as well as the shapes of each element and the purpose behind that element.

Then we select the ones that serve the diagram's purpose and skip the ones that do not offer any meaningful information. We will track all changes of the selected elements. The selection is based on:

- Serving the meaning of the UML diagram.
- No optional elements.

For example, in the UML class diagram, there is an element called comment. This element doesn't provide any structural meaning, so we are going to skip it. We identify the element's meaning in order to decide whether we have to include them in our assessment or not.

The last step in the analysis is selecting the UML diagram identifier. This identifier enables us to track the changes from one version to another. It is the identifier of the elements which we are going to compare.

4.2 Evaluation

Then we apply our assessment approach called the **Client Master Approach**. This approach is used to precisely track changes in relationships.

One of the main issues in following changes is avoiding computing the changes more than once, especially when the element has many relationships. This approach is used to indicate the **Client** side and the **Master** side of the relationship. The client element is the one that depends upon others, and will be affected by them. The master element is a standalone element and is not affected by others. The purpose behind this approach is to avoid duplication or counting change twice. The change counts for the client element.

For example, in the UML class diagram, if we take two classifiers *A* and *B* with an inheritance relationship, *B* inherits *A*. If this relationship is changed or deleted, for example, what is the real effect that happened, and which classifier is changed and which one is unchanged? Based on our approach, *B* represents the client side of the relationship, and *A* is the master one. So if the deletion happened then *B* is the affected element, and change is counted for it. *A* remains unchanged.

Based on this approach, we get all possible change combinations and detect which UML diagram element is changed, and which one remains unchanged.

4.3 Propose the Metrics

Next, we propose a metric for the UML class diagram, the UML use case diagram, and the UML sequence diagram. This is based on the assessment results, and the selected identifier for each one of them. To measure a UML diagram stability we handled each element's property separately and looked for the change that happened to the base version.

CHAPTER 5

STRUCTURAL STABILITY

This chapter explains the evaluation and assessment of the UML class diagram, and presents its structural stability metric.

5.1 Assessment

A class diagram is made up of a collection of classifiers and the relationships among them. We will apply the Client Master Approach to track changes. The classifier name will be used as the identifier. A class diagram is used to express the system structure; therefore we need to track all changes that may have any effect on the structure.

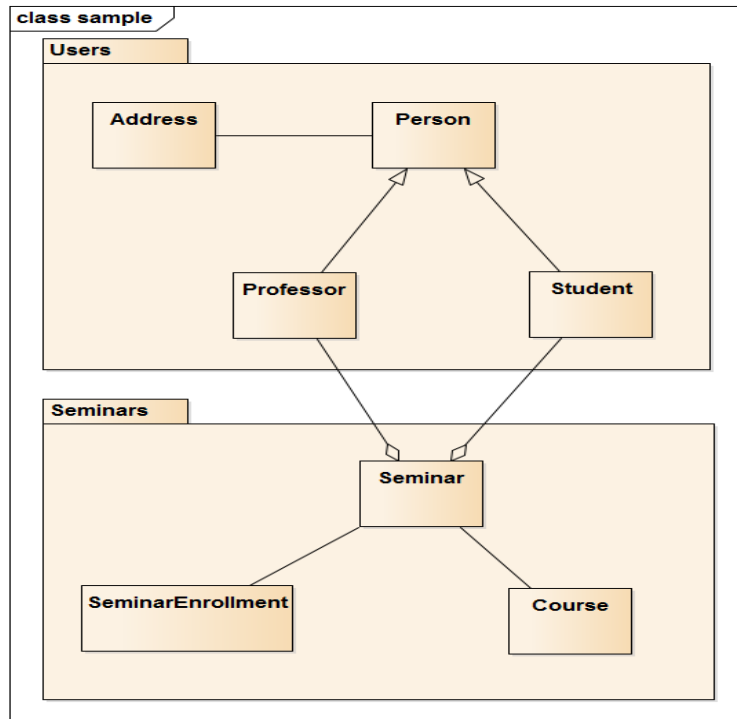


Figure 6 Class Diagram Sample

5.1.1 Classifiers

Figure 7 shows an example of a classifier with the detailed design. As we mentioned earlier, there are three blocks: classifier name, attributes, and operations. A classifier is identified by its name. The name is the inference of the classifier existence. If version *i* has class name *A*, then in the next version there will only be two cases; either the class still exists or it is removed. We do not capture the change in the class name. Renaming is not counted, because we cannot make sure that the class is renamed. The process to ensure that is to look into the methods and the attributes to determine if the classifier remains unchanged, or at least has the most attributes and methods. This process is not offered; next we will explain why that is.

Usually designers use class as a word, and not classifier. But here we use classifiers as a general word, because we deal with two types of them: the usual class and the interface. We are tracking a classifier type's change, so in order to avoid misunderstanding we chose to use classifier as a word.

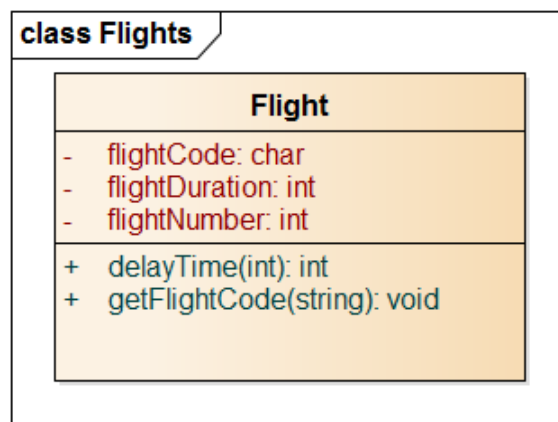


Figure 7 Detailed version of the class in class diagram

Methods, variables, and parameters can give us more details about the system. However, the detailed version of the classifiers in the class diagram is not always available. The second and third parts of the classifier are optional sections, as shown in Figure 8. System designers can hide these

sections. If these sections are not shown, it does not necessarily imply that they are empty, but that the diagram is perhaps easier to understand with that information hidden. Thus, we cannot ensure that these parts are hidden or do not exist at all. Therefore, we are going to skip them to be consistent with all diagram elements. We will not track the changes that may happen to these properties.

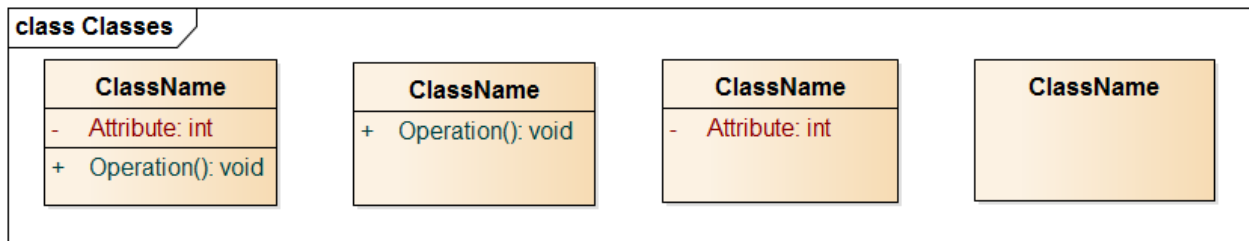


Figure 8 Four different ways of showing a class using UML notation

5.1.2 Comments

Comment shapes are used to annotate class diagrams. Comment shapes exist only on the diagram surface; they do not represent any structure and do not add any meaning to the class diagram. From its name it is just a comment to describe a class diagram and cannot exist in the code. There is no need to evaluate them, as they do not have any effect on the class diagram.

5.1.3 Packages

Grouping classifiers in packages gives the meaning of the organization. Packages exist to manage the large systems by dividing them into a group of classifiers. Usually package classifiers represent a specific part of the system. But this does not mean there is a change in class diagram structure before and after using packages. For example, in Figure 6, moving the *Seminar* classifier from the

Seminars package to the *User* package will not change the class diagram structure. Therefore there is no need to evaluate the classifier according to the package name. And when we are going to compute class diagram stability, we will deal with package content normally, as if there are no packages.

5.1.4 Dependency Relationship

Dependency relationship declares that one classifier depends upon another classifier. Figure 9 shows a sample diagram of a dependency relationship. The classifier *MenuItem* and the classifier *OrderItem* have a dependency relationship, if an object of one classifier might use an object of another classifier in the method definition. In this case, *OrderItem* uses *MenuItem* objects. The sample shows that the client classifier in this relationship is *OrderItem*, because it depends on *MenuItem*. And the master classifier here is *MenuItem*. *MenuItem* is a standalone classifier and does not depend on any other classifiers. Hence any change that may happen to the *MenuItem* will have a direct effect on *OrderItem*.

Note: in a dependency relationship, the client classifier is the one that depends on the other classifier.

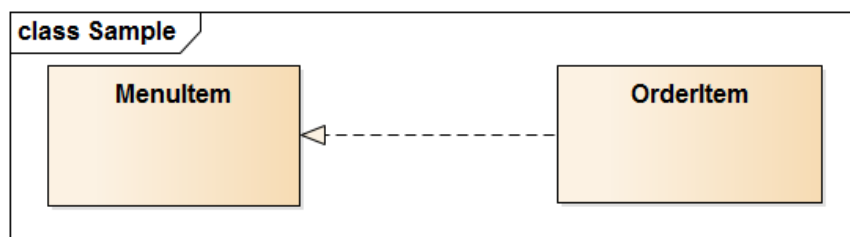


Figure 9 Class Diagram Dependency Relationship

Figure 10 explains how we can convert this type of relationship into a code. The code shows that *MenuItem* is Master because it is a standalone classifier. *OrderItem* uses *MenuItem* as a method parameter, as a method return type, or as a local variable. So it is a client in this case.

```
Class MenuItem{
}

Class OrderItem{
    Void method1(MenuItem parameter1){
    }

    MenuItem method2(){
    }





    void method3(){
        MenuItem variable1;
    }
}
```

Figure 10 Sample Code of Dependency Relationship

Table 3 shows all possible changes that may happen to the dependency relationship and their influence. The most affected one is classifier *B*, in almost all cases. According to the approach, *A* is the master and *B* is the client. *A* will be affected if it is changed to depend on *B*, and this happens in cases of a dependency relationship, an association relationship, if it aggregates or composes *B*'s objects, or inherits or realizes *B*.

For B, and because it is a client side, it is affected if any change happens to A, such as if A is deleted or renamed, the relationship is deleted, or the relationship is changed to any type or any direction.

Table 3 Dependency Relationship Possible Changes

Change Type	The Affected Classifiers	Justification
Rename classifier A	Classifier B	A will be counted as a new classifier because of renaming; therefore B will depend on the new classifier.
Delete classifier A	Classifier B	B depends on A, so because of deletion, the relationship will be deleted, and B will not depend on A.
Rename classifier B	-	A does not depend on any classifier, so the renaming of B will have no effect on A.
Delete classifier B	-	A does not depend on any classifier, so the deletion of B will have no effect on A.
Delete Relationship	Classifier B	A does not depend on B; however B depends on A. Therefore, because of this deletion, B will not depend on A.
	Classifier A, Classifier B	A does not depend on any classifier; however B depends on A. Because of the change, this is reversed.
	Classifier A, Classifier B	A does not depend on any classifier; it will be changed to use B's objects. In turn, B will use A's objects.
	Classifier A, Classifier B	A does not depend on any classifier; it will be changed to use B's objects. B will not depend on A.
	Classifier B	A does not depend on B; however, B depends on A, and it will be changed to use A's objects.

	Classifier A, Classifier B	A does not depend on any classifier; it will be changed to use B's objects. B will not depend on A.
	Classifier B	A does not depend on B. However, B depends on A, and it will be changed to use A's objects.
	Classifier B	A does not depend on B. However, B depends on A, and it will be changed to inherit A.
	Classifier A, Classifier B	A does not depend on any classifier; it will be changed to inherit B. B will be changed to realize A.
	Classifier A, Classifier B	A will be changed to an interface. B will be changed to realize A
	Classifier A, Classifier B	A does not depend on any classifier; it will be changed to realize B. B will be changed to an interface.

5.1.5 Association Relationship

An association relationship declares that objects of each classifier depend upon the objects of the other. Figure 11 shows a sample diagram of an association relationship. The classifier *Order* and the classifier *Customer* have an association relationship, if an object of one classifier might use an object of another classifier as a variable. In an association relationship case, the two classifiers are clients and masters at the same time. *Order* is a Master because *Customer* uses its objects. It is a Client because it depends on *Customer* classifier objects. The same reasoning applies to the *Customer*. Any change that may happen to the *Order* will have a direct effect on the *Customer*, and vice versa.

Note: in an association relationship, the two classifiers are clients and masters at the same time.

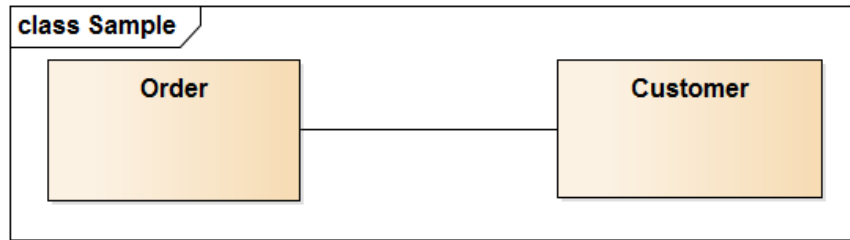


Figure 11 Class Diagram Association Relationship

Figure 12 explains how we can convert this type of relationship into a code. Classifiers have a reference to an object of the other classifiers as attributes.

```
public class Order {
    private Customer[] var1;
}

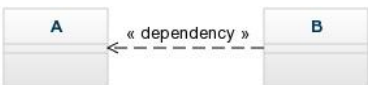
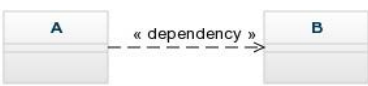
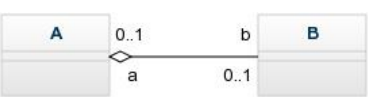

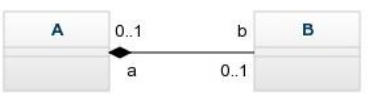
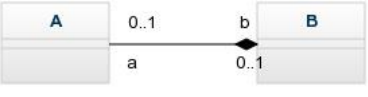

public class Customer {
    private Order[] var1;
}
```




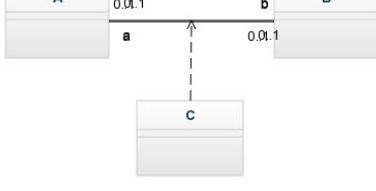
Figure 12 Sample Code of Association Relationship

Table 4 shows all change possibilities of this relationship and their influence. We can notice that the change of one classifier will affect the other, but the change that may happen to the relationship will affect the two classifiers.

Based on the approach, the two classifiers are clients. A will be affected if the relationship is changed to a dependency, an inheritance or realization, and if B aggregates or composes A's objects. B will be affected if the relationship is changed to dependency, inheritance, or realization, and if A aggregates or composes its objects.

Table 4 Association Relationship Possible Changes

Change Type	The Affected Classifiers	Justification
Rename classifier A	Classifier B	A will be counted as a new classifier because of renaming. Therefore B will use a new classifier's objects.
Delete classifier A	Classifier B	B is using A's objects, so because of deletion, the relationship will be deleted, and B will not use A's objects.
Rename classifier B	Classifier A	B will be counted as a new classifier because of renaming; therefore A will use a new classifier's objects.
Delete classifier B	Classifier A	A is using B's objects, so because of deletion, the relationship will be deleted, and A will not use B's object.
Delete Relationship	Classifier A, Classifier B	A does not depend on B; however B depends on A. Therefore, because of this deletion, B will not depend on A.
	Classifier A, Classifier B	A is using B's objects, but it will not use them after the change. B will be changed to depend on A.
	Classifier A, Classifier B	B is using A's objects, but it will not use them after the change. A will be changed to depend on B.
	Classifier B	A will remain using B's objects; however, B is changed, and it will not use A's objects.
	Classifier A	B will remain using A's objects; however, A is changed, and it will not use B's objects.
	Classifier B	A will remain using B's objects; however, B is changed, and it will not use A's objects.
	Classifier A	B will remain using A's objects; however, A is changed, and it will not use B's objects.
	Classifier A, Classifier B	A is using B's objects, but it will not use them after the change. However, B is using A's objects, and it will be changed to inherit A.

	Classifier A, Classifier B	B is using A's objects, but it will not use them after the change. However, A is using B's objects, and it will be changed to inherit B.
	Classifier A, Classifier B	A is using B's objects, but it will not use them after the change. However, B is using A's objects, and it will be changed to realize A.
	Classifier A, Classifier B	B is using A's objects, but it will be changed to an interface. However, A is using B's objects, and it will be changed to realize B.
	Classifier A, Classifier B	A will be changed to use C's objects, while B will not use A's objects.

5.1.6 Aggregation Relationship

An aggregation relationship declares that one classifier (a classifier with the diamond edge) is aggregated by the other objects. Figure 13 shows a sample diagram of an aggregation relationship. Classifier *Car* and classifier *Wheel* have an aggregation relationship if an object of *Car* aggregates an object of *Wheel*. Based on the Client Master Approach, classifier *Car* (whole) is the client, and classifier *Wheel* (part) is the master. *Car* is the client because it uses *Wheel* objects. *Wheel* is a master class, because it does not depend on *Car*. The existence of *Wheel* does not depend on the classifier *Car*. So if any change were happen to the *Wheel*, this would affect the *Car*.

Note: in an aggregation relationship the client classifier is the one that aggregates the other object; the other classifier is the master.

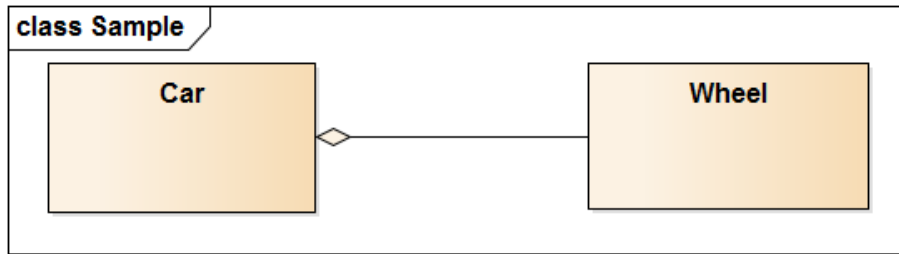


Figure 13 Class Diagram Aggregation Relationship

Figure 14 explains how we can convert this type of relationship into a code. Classifier *Car* objects aggregate classifier *Wheel* objects; in other words, *Car* owns *Wheel* objects. This is a one-way association so that the implementation code is like the implementation code of the association relationship.

```

public class Car {
    private wheel[] var1;
}

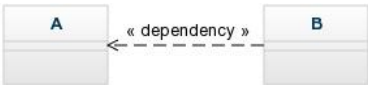
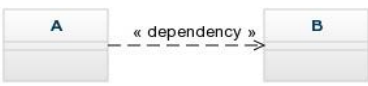



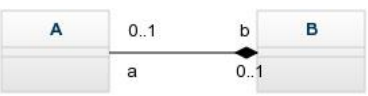

public class Wheel {
}
  
```

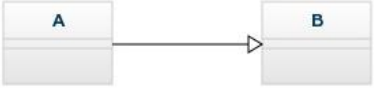


Figure 14 Sample Code of Aggregation Relationship

Table 5 shows all change possibilities of this relationship and their influence. According to the approach, *A* is the client, and *B* is the master. *A* will be affected in all change possibilities except that the relationship is changed to composition.

For *B*, and because it is in a master role, it is affected in the case that it uses *A*'s objects. This happens if the relationship is changed to a dependency, an inheritance, or realizations, and it aggregates or composes *A*'s objects.

Table 5 Aggregation Relationship Possible Changes

Change Type	The Affected Classifiers	Justification
Rename classifier A	-	B is not depending on A, so renaming A will not affect B,
Delete classifier A	-	B is not depending on A; therefore, deleting A will not affect B.
Rename classifier B	Classifier A	B will be counted as a new classifier because of renaming. Therefore, A will use a new classifier's objects.
Delete classifier B	Classifier A	A is using B's objects, so because of deletion, the relationship will be deleted, and A will not use B's objects.
Delete Relationship	Classifier A	B does not depend on A. However, A depends on B; therefore, because of this deletion, A will not depend on B.
	Classifier A, Classifier B	A is using B's objects, but it will not use them after the change. B will be changed to depend on A.
	Classifier A	A is using B's objects, but it will not use them after the change. No change happens to B; it still be a standalone.
	Classifier B	A will remain using B's objects; however B is changed, and it will use A's objects.
	Classifier A, Classifier B	A depends on B; however B is not depending on A. Therefore, because of the change, this is reversed.
	-	A will remain using B's objects; B will remain without any change.
	Classifier A, Classifier B	A depends on B; however B is not depending on A. Therefore, because of the change, this is reversed.
	Classifier A, Classifier B	A is using B's objects, but it will not use them after the change. However, B is not using A's objects, and it will be changed to inherit A.

	Classifier A	A is using B's objects, but it will be changed to inherit B after the change. However, B is not using A's objects, so it will remain without any change.
	Classifier A, Classifier B	A is using B's objects, but it will be changed to an interface. However, B is not using A's objects, and it will be changed to realize A.
	Classifier A, Classifier B	B is using A's objects, but it will be changed to an interface. However, A is not using B's objects, and it will be changed to realize B.

5.1.7 Composition Relationship

Composition declares that one classifier (a classifier with the diamond edge) composes the other objects. A composition is a stronger version of aggregation. Figure 15 shows a sample diagram of a composition relationship. Classifier *Person* and classifier *Hand* have a composition relationship, if *Person* aggregates the *Hand* object, and *Hand* objects cannot be aggregated by other classifiers than *Person* objects. The two classifiers are clients in this case. *Person* (whole classifier) is a client because it uses *Hand* objects. Also, *Hand* (part classifier) is a client classifier because its existence depends on *Person*. The *Hand* is actually part of *Person* itself and will not usually be shared with other parts of the class diagram. So if *Person* is deleted, then its corresponding parts are also deleted. Any change happening to one of them would affect the other one.

Note: in a composition relationship, the two classifiers are clients and masters at the same time.

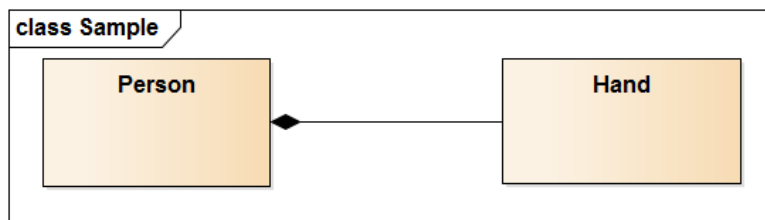


Figure 15 Class Diagram Composition Relationship

Figure 16 explains how we can convert this type of relationships into a code, the same as an aggregation relationship. The main difference in this case is that *Hand's* existence depends on *Person's* existence. It is also a one-way association, so that the implementation code is like the implementation code of the association relationship.

```

public class Person {
    private Hand[] var1;
}

public class Hand {
}

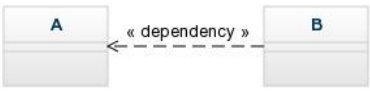
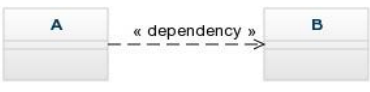

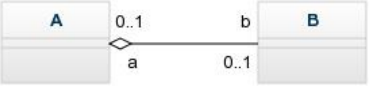
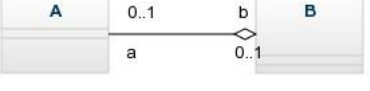


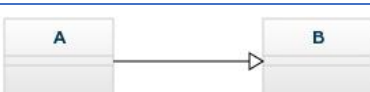


```

Figure 16 Composition Relationship Implementation Code

Table 6 tracks all possible changes of this relationship. Based on the client master approach. *A* is the client, and *B* is the master. *A* will be affected in all change possibilities except that the relationship is changed to aggregation. For *B*, and because it is in a master role, it is affected in the case that it uses *A's* objects. This happens if the relationship is changed to a dependency, inheritance, or realizations, and aggregates or composes *A's* objects.

Table 6 Composition Relationship Possible Changes

Change Type	The Affected Classifiers	Justification
Rename classifier A	-	B is not depending on A, so renaming A will not affect B
Delete classifier A	Classifier B Deleted	B is not depending on A. However, B's existence depends on A; therefore deleting A will lead to B's deletion.
Rename classifier B	Classifier A	B will be counted as a new classifier because of renaming; therefore A will use a new classifier's objects.

Delete classifier B	Classifier A	A is using B's objects, so because of deletion, the relationship will be deleted, and A will not use B's objects.
Delete Relationship	Classifier A, Classifier B Deleted	A is using B's objects, so because of deletion, A will not use B's objects. However, B's existence depends on A, therefore deleting A will lead to B's deletion.
	Classifier A, Classifier B	A is using B's objects, but it will not use them after the change. B will be changed to depend on A.
	Classifier A	A is using B's objects, but it will not use them after the change. No change happens to B; it is still a standalone.
	Classifier B	A will remain using B's objects; however B is changed, and it will use A's objects.
	-	A will remain using B's objects; B will remain without any change.
	Classifier A, Classifier B	A depends on B; however, B is not depending on A. Therefore, because of the change, this is reversed.
	Classifier A, Classifier B	A depends on B; however B is not depending on A. Therefore, because of the change, this is reversed.
	Classifier A, Classifier B	A is using B's objects, but it will not use them after the change. However, B is not using A's objects, and it will be changed to inherit A.
	Classifier A	A is using B's objects, but it will changed to inherit B after the change. However, B is not using A's objects, so it will remain without any change,
	Classifier A, Classifier B	A is using B's objects, but it will changed to an interface. However, B is not using A's objects, and it will be changed to realize A.
	Classifier A, Classifier B	B is using A's objects, but it will changed to an interface. However, A is not using B's objects, and it will be changed to realize B.

5.1.8 Inheritance Relationship

An inheritance relationship means that one classifier is a type of another one. Figure 17 shows a sample diagram of an inheritance relationship. The *Person* classifier and the *Student* classifier have an inheritance relationship, if one classifier is a type of the other. Based on the Client Master Approach, *Person* is the master classifier, and *Student* is the client. The *Student* is a client classifier, because it is depend on *Person*. Therefore, if any change happens to the *Person* classifier, this might affect the *Student*.

Note: in an inheritance relationship, the super classifier is the master, and the sub-classifier is the client.

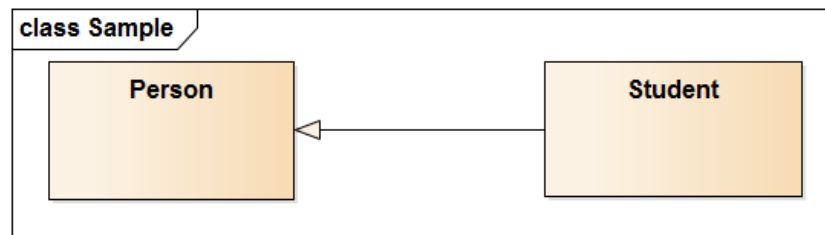


Figure 17 Class Diagram Inheritance Relationship

Figure 18 explains how we can transform this type of relationships into a code. *Student* extends class *Person*. In other words, *Student* is a type of *Person* and inherits all its attributes and methods that are declared.

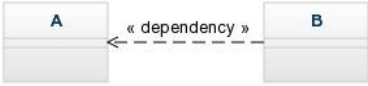
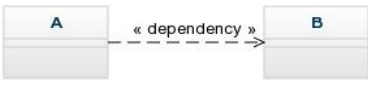


```
public class Person {
}







public class Student extends Person{
}
```

Figure 18 Inheritance Relationship Implementation Code

Table 7 shows all change possibilities of this relationship and their influence. According to the approach, *A* is the master and *B* is the client. *A* will be affected if it is changed to depend on *B*, and this happens in cases of a dependency relationship or an association relationship, aggregates or composes *B*'s objects, or inherits or realizes *B*. For *B*, and because it is in a client role, it is affected if any kind of changes happen.

Table 7 Inheritance Relationship Possible Changes

Change Type	The Affected Classifiers	Justification
Rename classifier A	Classifier B	A will be counted as a new classifier because of renaming; therefore B will inherit a new classifier.
Delete classifier A	Classifier B	B is inheriting A, so because of deletion, the relationship will be deleted, and B will not inherit A.
Rename classifier B	-	A does not depend on any classifier, so the renaming of B have no effect on A
Delete classifier B	-	A does not depend on any classifier, so the deletion of B will have no effect on A.
Delete Relationship	Classifier B	A does not depend on B; however B is inheriting A. Therefore, because of this deletion, B will not inherit A.
	Classifier B	A does not depend on any classifier; however B is inheriting A. Because of the change, B will depend on A.
	Classifier A, Classifier B	A does not depend on any classifier, but it will be changed to depend on B. After the change B will not inherit A.
	Classifier A, Classifier B	A does not depend on any classifier, but it will be changed to use B's objects. B will be changed to use A's objects.
	Classifier A, Classifier B	A does not depend on any classifier, but it will be changed to use B's objects. However, B will not inherit A.

	Classifier B	A does not depend on any classifier. B will be changed to use A's objects.
	Classifier A, Classifier B	A does not depend on any classifier, but it will be changed to use B's objects. However, B will not inherit A.
	Classifier B	A does not depend on any classifier. B will be changed to use A's objects.
	Classifier A, Classifier B	A will be changed to inherit B. B will not inherit any classifier.
	Classifier A, Classifier B	A will be changed to an interface. B will be changed to realize A.
	Classifier A, Classifier B	A does not inherit any classifier, but it will be changed to realize B. B will be changed to an interface.

5.1.9 Realization Relationship

A realization relationship means that one classifier is realized by another one. Figure 19 shows a sample diagram of a realization relationship. Classifier *Service* and classifier *Customer* have a realization relationship, if one classifier has implemented the other classifier's methods. Using the Client Master Approach, the master classifier is the one that others realize (a classifier with an arrowhead), which is called the interface classifier. The other one is the client. Any change happening to the master classifier would affect the other one. In our sample, *Service* is the master, and *Customer* is the client. *Customer* is a client because it realizes the *Service* classifier. So if any change happens to the *Service*, this might affect the *Customer*.

Note: in a realization relationship, the client classifier is the one that realizes the other object. The other one is the master.

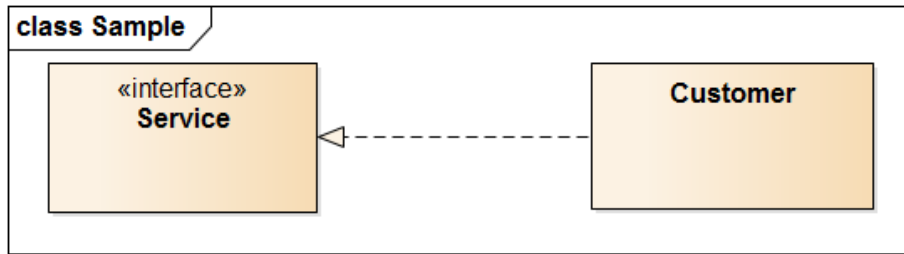


Figure 19 Class Diagram Realization Relationship

Figure 20 explains how we can convert this type of relationships into a code. *Service* implements the *Customer*. In other words *Service* implements all the methods that are declared in the *Customer* classifier.

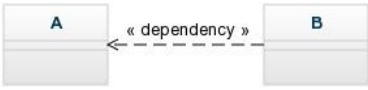
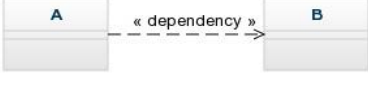

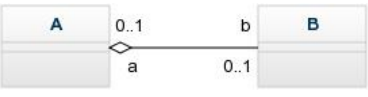

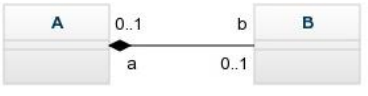
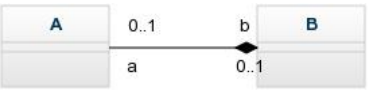
```
public class Service {
}


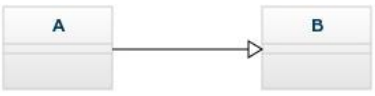

public class Customer implements Service{
}
```

Figure 20 Realization Implementation Code

Table 8 shows all change possibilities of this relationship and their influence. Based on the approach, *A* is the master and *B* is the client. *A* will be affected if it is changed to depend on *B*, and this happens in cases of a dependency relationship, or an association relationship, aggregates or composes *B*'s objects, or inherits or realizes *B*. *B* it is affected if any kind of change is happening.

Table 8 Realization Relationship Possible Changes

Change Type	The Affected Classifiers	Justification
Rename classifier A	Classifier B	A will be counted as a new classifier because of renaming. Therefore B will realize a new classifier.
Delete classifier A	Classifier B	B realizes A, so because of deletion, the relationship will be deleted, and B will not realize A.
Rename classifier B	-	A does not depend on any classifier, so the renaming of B will have no effect on A
Delete classifier B	-	A does not depend on any classifier, so the deletion of B will have no effect on A
Delete Relationship	Classifier B	A does not depend on B; however B realizes A. Therefore, because of this deletion, B will not realize A.
	Classifier B	A does not depend on any classifier; however B realizes A. Because of the change, B will depend on A.
	Classifier A, Classifier B	A does not depend on any classifier, but it will be changed to depend on B. After the change B will not realize A.
	Classifier A, Classifier B	A does not depend on any classifier, but it will be changed to use B's objects. B will be changed to use A's objects.
	Classifier A, Classifier B	A does not depend on any classifier, but it will be changed to use B's objects. However, B will not realize A.
	Classifier B	A does not depend on any classifier. B will be changed to use A's objects.
	Classifier A, Classifier B	A does not depend on any classifier, but it will be changed to use B's objects. However, B will not realize A.
	Classifier B	A does not depend on any classifier. B will be changed to use A's objects.

	Classifier A, Classifier B	B will be changed to inherit A. A will be changed to a class.
	Classifier A, Classifier B	A does not inherit any classifier, but it will be changed to inherit B. B will not inherit any classifier.
	Classifier A, Classifier B	B will be changed to an interface. A will be changed to realize B.

5.1.10 Association classes

These classes are new classes. They can be introduced by the association itself. Association classes are particularly useful in complex cases when you want to show that a class is related to two classifiers because those two classifiers have a relationship with each other. In Figure 21, the association relationship between *Student* and *Course* results in an association relationship with a set of objects in classifier *Enrollment*. Based on the approach, *Student* and *Course* are still master and clients at the same time. For the new classifier it is also a client and a master, *Student* and *Course* used it, and its existence depends on the relationship.

Note: in association classes, all the partners are clients and masters at the same time.

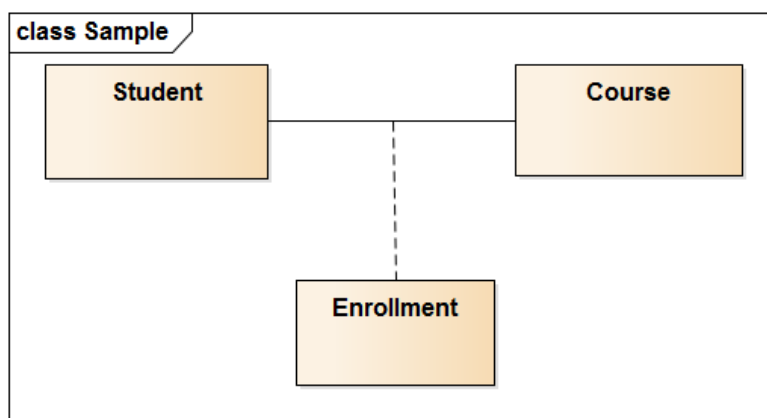


Figure 21 Association Class in Class Diagram

Figure 22 explains how we can transform this type of relationship into a code. Classifiers have a reference to an object of the other classifier as an attribute.

```

public class Student {
    private Course[] var1;
    private Enrollment[] var2;
}

public class Enrollment {
}

public class Course {
    private Enrollment [] var1;
}


```

Figure 22 Association Class Implementation Code

Table 9 shows all change possibilities of this relationship and their influence. Classifier A is affected by this type of relationships if B is changed, deleted or renamed. And if an association class is emerging. B is affected in the case of, delete A, or delete C, or an association class is emerging.

Table 9 Association Class Possible Changes

Change Type	The Affected Classifiers	Justification
Rename classifier A	-	A will be counted as a new classifier because of renaming. However, B does not depend on any classifier.
Delete classifier A	Classifier B, Classifier C Deleted	B is using A's objects, so because of deletion, the relationship will be deleted, and B will not use A's objects. C also depends on the relationship, so it will be counted as deleted.

Rename classifier B	Class A	B will be counted as a new classifier because of renaming; therefore A will use a new classifier's objects.
Delete classifier B	Classifier A, Classifier C Deleted	A is using B's objects, so because of deletion, the relationship will be deleted, and A will not use B's objects. C also depends on the relationship, so it will be counted as deleted.
	Classifier B	A will use B's objects, and B will use A's objects.

5.1.11 The Selected UML Class Diagram Identifier

Here we selected the classifier name as an identifier, as it is the most appropriate property. The possible changes in the identifier are deletion and renaming. Renaming, as we mentioned earlier, cannot be detected. So we will deal with the unchanged aspect only.

5.1.12 Summary

The following are the list of all selected elements and attributes, which we are going to evaluate and track their unchanged:

- Classifiers name.
- Classifiers type.
- Dependency relationship.
- Association relationship.
- Aggregation relationship.
- Composition relationship.
- Inheritance relationship.
- Realization relationship.

5.2 Terminology and Formalism

In this section we will identify the terminology and formalism used during stability computation.

Definition 1 (CLASSIFIER). Let the class diagram classifiers be denoted by C . The same classifier can have different versions based on different class diagram versions. Let C_i denote the classifier C in the class diagram version i , where $i \in [1..n]$.

Definition 2 (CLASSIFIER PROPERTIES). Let $P(C_i)$ denote the set of all properties of the classifier C in the class diagram version i .

Definition 3 (CLASSIFIER TYPE). Let the classifier type in the class diagram be denoted by CT . The same classifier can have different values based on different class diagram versions.

Definition 4 (NUMBER OF CLASSIFIERS OF CLASS DIAGRAM BASE VERSION). Let NC represent the number of classifiers in the class diagram base version.

Definition 5 (CLASSIFIER PROPERTIES CHANGE). Let changes that may happen to any classifier be denoted by Ch . Ch represents any change in classifier properties from the class diagram base version to any other class diagram version.

Definition 6 (CLASSIFIER CHANGES)

CC is the percentage of class diagram classifier changes.

Definition 7 (NUMBER OF UNIQUE PROPERTIES). The classifier in the class diagram has different properties. These properties represent the classifier type and its classifier relationships with other classifiers. Let the number of unique properties be denoted by NUP .

Definition 8 (CLASSIFIER RELATIONSHIPS).

Let classifier relationships be denoted by CR.

Definition 9 (NUMBER OF UNIQUE CLASSIFIER RELATIONSHIPS). Let the number of unique classifier relationships in the class diagram classifier be denoted by NUCR.

Definition 10 (STRUCTURAL STABILITY)

SS is the percentage of structural stability.

5.3 Structural Stability Metric

To measure class diagram stability we handled each classifier property separately and looked for the change in the base version. Figure 23 summarize the computation steps. The measurement of the class diagram stability is done through the following steps:

1. Develop a property change metric, a metric that measures the changes of each classifier property. Property change is computed according to Figure 24, and Figure 25, which illustrates classifier type changes and classifier relationship changes.
2. Count the summation of all property change metrics and divide them by the number of classifier unique properties, Equation 4.1. Unique properties are the number of unique classifier relationships plus one (one denoted for classifier type). Dividing by the number of unique classifier properties will normalize the classifier changes result to be between zero and one. One means all classifier properties have been changed from the i version to the $i+1$ version.
3. Get the summation all the classifiers change metrics and divide them by the number of class diagram base version classifiers. Dividing by the number of base version classifiers

will normalize the result to be between zero and one. One means all class diagram classifiers have been changed from the i version to the $i+1$ version.

4. The overall class diagram stability metric is computed using Equation 4.2. The final value is also normalized. Zero means all classifiers have been changed from version i to version $i+1$. Thus, version $i+1$ is unstable. On the other hand, one means nothing has been changed. Therefore, version $i+1$ is completely stable.

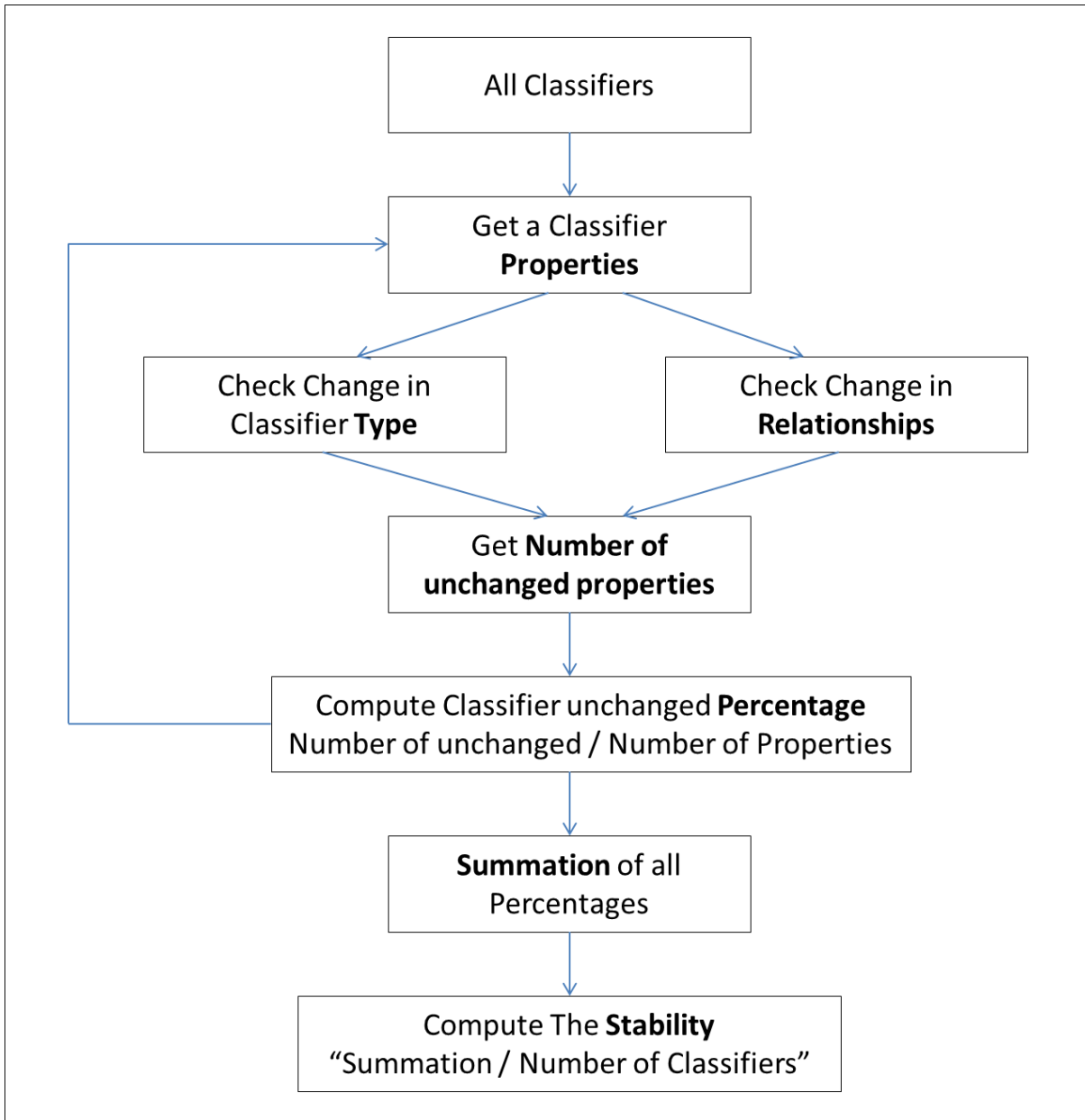


Figure 23 Structural Stability Computation Steps

To count the changes that may happen to the classifier properties we have to check first if the classifier is still in version $i+1$ or not. This is done using the selected identifier. If the identifier were deleted, then the classifier change value will be the maximum value, one. Otherwise, we will compute each classifier property change according to Figure 24 and Figure 25.

Figure 24 represents classifier type changes. Zero means the classifier type remains unchanged. One means that the classifier type is changed, either from class to interface, or from interface to class.

$$Ch(CT) = \begin{cases} 0, & \text{Class Type Changed} \\ 1, & \text{Class Type Unchanged} \end{cases}$$

Figure 24 Classifier Type Changes

Figure 25 represents classifier relationship changes. The change counts as one in two cases: if the relationship is deleted or if it is changed to another type. The change will be zero if the relationship remains unchanged.

$$Ch(CR) = \begin{cases} 0, & \text{Relationship Deleted} \\ 0, & \text{Change Relationship Type} \\ 1, & \text{Relationship Unchanged} \end{cases}$$

Figure 25 Classifier Relationships Changes

$$UCC = \frac{Ch(CT(i,i+1)) + \sum_{CR=1}^{NUCR} Ch(CR(i,i+1))}{NUP} \quad 4.1$$

UCC is an abbreviation for Unchanged in Classifier. This metric computes the unchanged of each classifier, which equals the summation of classifier type changes and all relationship changes over the number of unique properties.

NUCR is the abbreviation for the Number of Unique Classifier Relationships in the class diagram classifier.

CR is the abbreviation for Classifier Relationship.

CT is the abbreviation for Classifier Type.

NUP is the abbreviation for Number of Unique Properties. $NUP = (NUCR + 1)$, where 1 represents the classifier type

i: a class diagram version

$$SS(i + 1) = \frac{\sum_{C=1}^{NC} UCC(C)}{NC} \quad 4.2$$

SS is an abbreviation for Structural Stability. This metric computes the stability of the class diagram, which equals the summation all classifiers' change over the number of base version classifiers, and the value is subtracted from one.

C is the abbreviation for Classifier.

CC is the abbreviation for Classifier Change.

NC is the abbreviation for Number of Classifiers in the base version.

The following example shows the steps to measure class diagram stability.

5.3.1 Example

Figure 26 shows version *i* of a sample class diagram, and Figure 27 shows version *i+1* of the same sample of a class diagram.

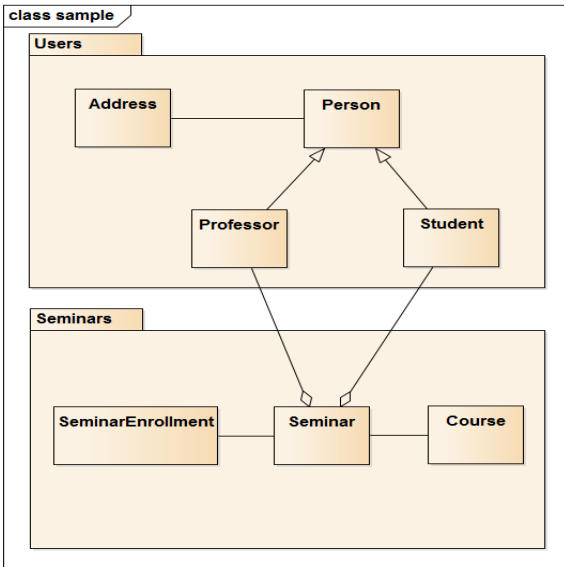


Figure 26 Class Diagram Sample version i

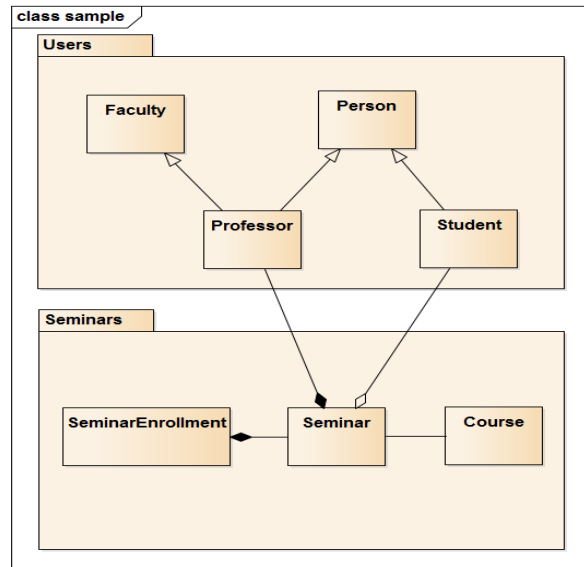


Figure 27 Class Diagram Sample version $i+1$

Table 10 shows the each classifier properties for each version. Table 11 shows the calculation of all the unchanged from sample class diagram version i to sample class diagram version $i+1$.

Table 10 Each Classifier Properties

Identifier	Properties	version i Data	version $i+1$ Data
Person	Classifier Type	Class	Class
	Relationships	Address-Association	-
Address	Classifier Type	Class	Deleted
	Relationships with	Person-Association	
Professor	Classifier Type	Class	Class
	Relationships	Person-Inheritance	Person-Inheritance
		-	Faculty-Inheritance
Student	Classifier Type	Class	Class
	Relationships	Person-Inheritance	Person-Inheritance
Seminar	Classifier Type	Class	Class
	Relationships	Professor-Aggregation Student-Aggregation	Professor-Composition Student-Aggregation

		Course-Association SeminarEnrollment- Association	Course-Association -
Course	Classifier Type	Class	Class
	Relationships	Seminar-Association	Seminar-Association
EnrollmentSeminar	Classifier Type	Class	Class
	Relationships	Seminar-Association	Seminar-Composition

Table 11 Changes From version i to version $i+1$

Identifier	Changes	No. of Unique Properties	Unchanged Value	Unchanged Average
Person	Address-Association DELETED	2	1	0.5
Address	CLASSIFIER DELETED	2	0	0
Professor	-	3	2	0.6
Student	-	2	2	1
Seminar	Professor[Aggregation => Composition] SeminarEnrollment-Association DELETED	5	3	0.6
Course	-	2	2	1
EnrollmentSeminar	Seminar[Association => Composition]	2	1	0.5

$$SS(i + 1) = \frac{0.5 + 0 + 0.6 + 1 + 0.6 + 1 + 0.5}{7}$$

$$SS(i + 1) = 0.6$$

The 0.6 means that version $i+1$ of the sample class diagram's stability is 60%; in other words version $i+1$ kept 60% of the version i structure, elements, and attributes. Sixty percent of classifiers and relationships remain in the next version.

CHAPTER 6

FUNCTIONAL STABILITY

This chapter presents the evaluation and assessment of the UML use case diagram, and presents its functional stability metric.

6.1 Assessment

The use case diagram is made up of a collection of actors, use cases, and relationships between and among them. Like the class diagram, we will apply the Client Master Approach to track changes. We will use either the actor name or the use case name as the identifier in the use case diagram.

A use case diagram is used to represent system functionality; therefore, we have to track all the properties that may affect the functionality of the system.

6.1.1 Actor

The actor is a part of the system functionality. It is the one which initiates the use case. An actor doesn't need to be a human user; any external system element outside of the use case may trigger the use case. The actor is not always used to trigger use case (send data); it can receive data also. The actor can have a relationship with another actor or with a use case. We selected the actor name as one of the identifiers.

From the actor information, only its name is involved in change assessment. The actor representation part is neglected; it does not have any functionality meaning. We can identify the actor by its name only, so that if the actor name is changed, we cannot recognize the original one. In this case we deal with the actor by considering that it is deleted and a new actor is emerging.

6.1.2 Use Case

Use case is used to represent the functionality of the system. It describes what a system does, but it does not specify how it does it. Use case typically represents a major piece of functionality; it is a description of a set of sequential actions, including variants that a system performs to yield an observable result to an actor.

Figure 28 shows a sample use case. The information that can be provided by the use case is the use case name, and the use case type. The use case name is used as a description of a functionality. In the sample, the use case name *RentACar* indicates a specific system functionality, which is a car rental. For the type, we mean whether it has an extension point or is just a normal use case. A use case with an extension point is used when the use case is extending another one. In the sample we have an *extension1* that shows the extension information with other use cases.

These two parts, the use case name and the use case type, are involved in the use case diagram assessment. We use the use case name as an identifier in comparison. Any change in the name cannot be recognized. For example, in our sample if the name is changed to *CarRental*, there are no indicators that they were the same functionality. So again we do not look at renaming, we deal with use case as either being added or deleted. We cannot ensure that the use case has been renamed.

Use case can be represented by different shapes; however, this representation is neglected and does not provide any functionality meaning.

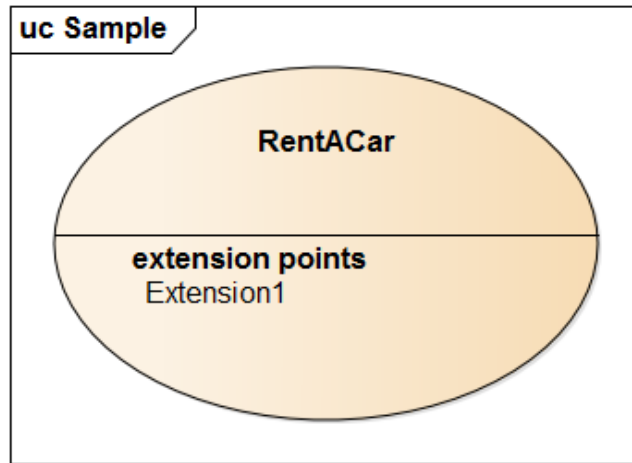


Figure 28 Sample Use Case with Extension Point

Table 12 shows the possible changes with the new relationships. *A* or *B* is affected if they included another use case, extend another use case, or use another use case as a general one.

Table 12 Possible Use Case Changes

Change Type	The Affected Use Cases	Justification
A includes B	Use Case A	A will be changed to depend B. B is a standalone use case.
B includes A	Use Case B	B will be changed to depend on A. A is a standalone use case.
A is an extension to B	Use Case B	A is a standalone. Therefore, because of the relationship, an extension point emerges in B.
B is an extension to A	Use Case A	B is standalone. Therefore, because of the relationship, an extension point emerges in A.
A is a generalization of B	Use Case B	B will be changed to depend on A. A is complete on its own.
B is a generalization of A	Use Case A	A will be changed to depend on B. B is complete on its own.
A is an association Actor	-	Use case A will not be changed.

6.1.3 System Boundaries

The use of the system boundaries is for the purposes of organization only. The boundaries are represented in a generic sense using a simple rectangle, with the name of the system at the top. Whether the designer decides to use it when he is designing the system or not will not change any system functionality. Therefore we are going to exclude it from our assessment.

6.1.4 Actor Relationships

The actor has two kinds of relationships, and one is a relationship with another actor as shown in Figure 29. The second, main one, is the relationship with the use cases, shown in Figure 30.

Actors can be generalized like many other classifiers. Actor generalization is typically used to pull out common requirements from several different actors to simplify modeling. Generalization is attained by creating a generic actor to capture the common functionality, and then specialized to identify the unique needs of each actor. The relationship can be represented by drawing a solid line, with a closed arrow pointing from the specialized actor to the base actor. In Figure 29, *Administrator* represents the master side of this relationship; any change which happens to this actor it will affect the *DBAdministrator*, which is a client in this relationship. The possible changes that may happen in this case are the deletion of the actor *Administrator*, or the deletion of the relationship itself. These changes will have a direct effect on the *DBAdministrator* actor. Another possible change is the deletion of the existing relationship, where a reverse relationship will emerge instead. In this case both actors are affected by this change.

Note: in an actor-actor relationship, the general actor is the master one, and the specialized actor is the client one.

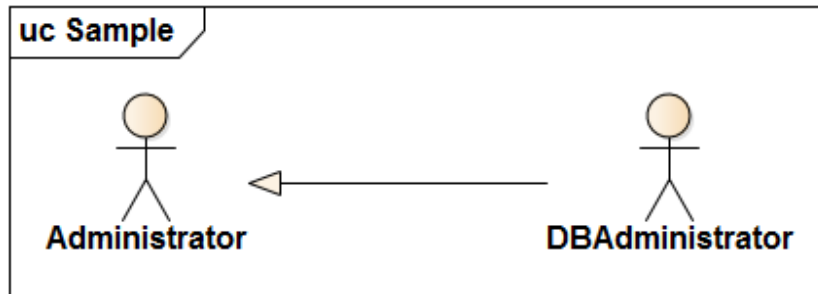


Figure 29 Sample Actor-Actor Relationship

The second relationship is with the use cases. The actor can be associated with one or more use cases. This relationship is represented by a solid line. A relationship between an actor and a use case indicates that the actor initiates the use case, or the use case provides the actor with results, or both. Figure 30 shows a sample relationship between an actor and a use case. In this relationship, both the *Driver* actor and *RentACar* use case are clients. For the *RentACar* use case, it represents a client role because its initiation is based on the actor, so changing the actor to another one will lead to a different functional meaning.

For the actor, as we mentioned earlier, the actor can be used to send data or receive data. However, sometimes we cannot ensure what the case is exactly, but as we have to consider the case when it receives data, we have to involve it in the assessment; this is why the actor represents a client role in this relationship.

Usually use cases are depicted in a standard way in drawing and reading, which is from left to right. The actors initiating use cases are on the left and actors that receive use case results are on the right. However, depending on the model or level of complexity, it may make sense to group actors differently. So we cannot rely on this tradition, it is not mandatory and depends on the system. Thus we cannot differentiate between the imitating actors and receiving actors. Therefore, we will deal with all actors as if they were clients when they communicate with use cases.

Note: in the actor use case relationship the use case and the actor are Clients.

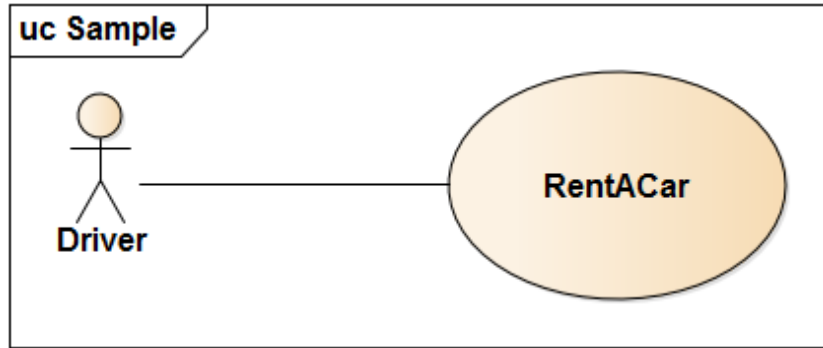



Figure 30 Sample Actor Use Case Relationship

Table 13, Table 14 shows all the possible changes in the actor-use case relationships, and actor-actor relationships. Using the client master approach, the *Actor* and *A* are clients. *A* will be affected if the actor is changed to another one, or deleted, or the relationship itself is deleted. The Actor is affected if *A* is changed to another use case, or it is deleted, or if it is generalized to another actor.

Table 13 Possible Actor-Use case Relationships Changes

Change Type	The Affected Entity	Justification
Change use case A to another one	Actor	Use case A will be counted as deleted; the Actor will depend on a new Use Case
Change the Actor to another one	Use Case A	A will depend on a new Actor because the Actor will be counted as deleted.
Delete use case A	Actor	When the use case is deleted, the Actor will depend on a new use case
Delete the Actor	Use Case A	A will depend on a new Actor because of deletion.
Delete the relationship	Use Case A, Actor	A will not depend on an Actor; An Actor will not depend on A.

Table 14 Possible Actor-Actor Relationships Changes

Change Type	The Affected Entity	Justification
Delete relationship	Actor2	Actor1 does not depend on Actor2; however Actor2 depends on Actor1. Therefore, because of this deletion, Actor2 will not depend on Actor1.
	Actor1, Actor2	Actor1 does not depend on any classifier; however, Actor2 depends on Actor1. Because of the change, this is reversed.

6.1.5 Generalization Relationship

A generalization relationship is used to express higher level functionality. The use case generalization can be represented using a solid line, with a closed arrow pointing from the specialized use case to the base use case. Figure 31 shows a sample diagram of this relationship. The use case *Authentication* represents the generic use case, while the use case *EmailLogin* represents a specialization of the use case *Authentication*. Even with the generalization, we are still talking about the system functionality, not an implementation, and hence the two use cases- the generic one and the specialized one- are involved the assessment. The generalization can also be called inheritance.

From the sample use case *Authentication* represents the master side of the relationship, because it contains general steps that can be used by the inherit use case. Now, if we want to access use case *EmailLogin*, we have to use the use case *Authentication* steps, because every step in the general use case *Authentication* must occur in the specialized use case *EmailLogin*. Therefore the use case *EmailLogin* represents the client side of the relationship.

Note: in a use case generalization relationship, the generic use case is the master one, and the specialization use case is the client one.

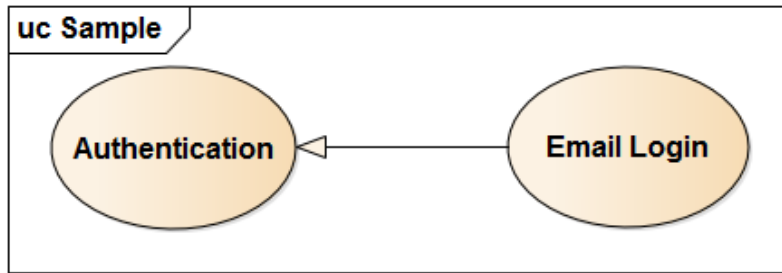
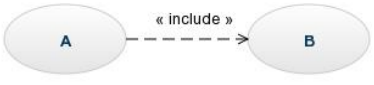
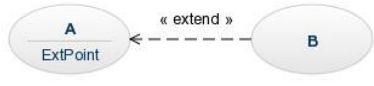
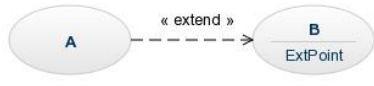



Figure 31 Use Case Generalization

The possible changes that may happen to the generalization relationship are shown in Table 15. Based on the approach, A is the master side of the relationship, and B is the client one. A will be affected if the relationship is changed to extend or include, whatever its direction. B is affected in all cases, except the case that is included A.

Table 15 Generalization Relationship Possible changes

Change Type	The Affected Use Cases	Justification
Change use case A to another one	Use Case B	A will be counted as a deleted use case; therefore B will depend on a new. one
Change use case B to another one	-	A does not depend on any classifier, so the deletion of B will have no effect on A
Delete use case A	Use Case B	B will be counted as a deleted use case. However, A is a standalone use case.
Delete use case B	-	A does not depend on any classifier, so the deletion of B will have no effect on A.
Delete Relationship	Use Case B	A does not depend on any use case. However, B does, so because of the change it will not depend on any use case.
	-	A does not depend on any use case, and will remain the same after the change. B remains the same; it depends on A.

	Use Case A, Use Case B	A does not depend on any use case, but it will be changed to depend on B. B will be changed from depending on A to a standalone use case.
	Use Case A, Use Case B	A will be changed to have an extension point. B's existence is dependent on A.
	Use Case A, Use Case B	A does not depend on any use case, but it will be changed to depend on B. B will be changed to have an extension point.
	Use Case A, Use Case B	A does not depend on any use case; however, B depends on A. Because of the change, this is reversed.

6.1.6 Include Relationship

An include relationship is used in the case of creating a shared and common functionality. The purpose of this action is behavior modularization, making them more manageable. The use case inclusion is represented using a dashed line, with an open arrow (dependency) pointing from the base use case to the included use case. The line is labeled with the keyword include. Figure 32 shows a sample diagram of the include relationship. Use case *OrderAMeal* represents the including use case, while use case *Pay* represents the included use case. An include relationship means that the behavior in the additional use case (*Pay*) is inserted into the behavior of the base use case (*OrderAMeal*).

From the sample, the *OrderAMeal* use case represents the client side of this relationship. In order to access or perform *OrderAMeal* we have to perform use case *Pay*, because *OrderAMeal* is not complete on its own. In other words, *OrderAMeal* depends and needs *Pay*. However, use case *Pay* can be complete, and can be accessed without the need of use case *OrderAMeal*; *Pay* represents the master side of this relationship.

Note: in an include relationship, the master is the including use case, and the client is the included use case.

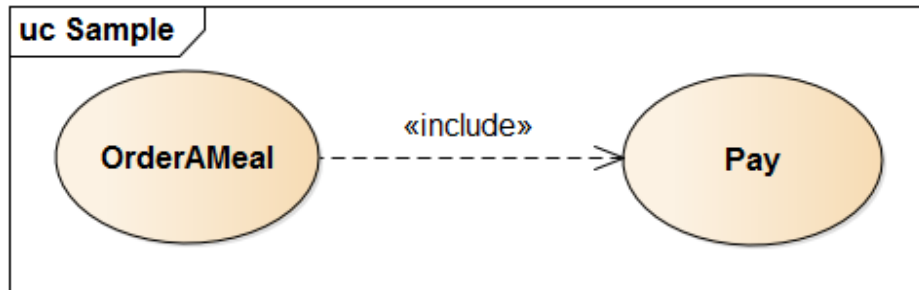
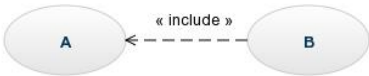
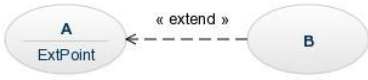
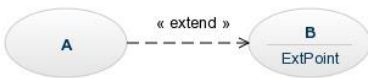




Figure 32 Use Case Inclusion Sample

The possible changes that may happen to the include relationship are shown in Table 16. According to the approach, A is the client side of the relationship, and B is the master. A will be affected if the relationship changes to any other type. B affected in all cases except in the case that A is used as a general use case.

Table 16 Include Relationship Possible changes

Change Type	The Affected Use Cases	Justification
Change use case A to another one	-	A will be counted as a deleted use case. However, B is a standalone use case.
Change use case B to another one	Use Case A	B will be counted as a deleted use case; therefore A will depend on a new one.
Delete use case A	-	A will be a deleted use case. However, B is a standalone use case
Delete use case B	Use Case A	A depends on B, so the deletion of B will have effect on A. A will not depend on any use case.
Delete Relationship	Use Case A	A depends on B, so the deletion of the relationship will have an effect on A. A will not depend on any use case.
	Use Case A, Use Case B	B does not depend on any use case; however A depends on B. Because of the change, this is reversed.

	Use Case A, Use Case B	B does not depend on any use case, but it will be changed to depend on A. A will be changed to have an extension point.
	Use Case A, Use Case B	B will be changed to have an extension point. A's existence is dependent on B.
	-	B does not depend on any use case, and remains the same after the change. A remains the same; it depends on B.
	Use Case A, Use Case B	B does not depend on any use case, but it will be changed to depend on A. A will be changed from depending on B to being a standalone use case.

6.1.7 Extend Relationship

An extend relationship is used to plug in additional functionality to the base use case. It defines that instances of a use case may be added with some additional functionality to an extended use case. Use case extension is represented using a dashed line, with an open arrow (a dependency) pointing from the extension use case to the base use case. The line is labeled with the keyword `« extend »`. Figure 33 shows a sample diagram of the extend relationship. *ViewAccountDetails* expresses the extended use case, while *ViewHistory* expresses the extending use case. The relationship in this example indicates that the *ViewHistory* inserts additional action sequences into the *ViewAccountDetails* sequence. This allows *ViewHistory* to continue the activity sequence of *ViewAccountDetails* when the appropriate extension point is reached in the *ViewAccountDetails*, and the extension condition is fulfilled. In other words, a *ViewHistory* use case continues the functionality of a *ViewAccountDetails* use case

Accordingly, *ViewAccountDetails* is an independent use case, hence it has to describe the master role of the relationship; however there is a point we cannot overlook, which is the extension point.

An extension point is a specification of some point in the use case where an extension use case can plug in and add functionality. UML doesn't have a particular syntax for extension points; they are typically freeform text. The extension point is introduced to the *ViewAccountDetailes* because of the extend relationship. In this case the owner and the controller of this relationship is *ViewHistory*. This happens in the case of deletion of the *ViewHistory*. Deletion of a *ViewHistory* will have a direct effect on *ViewAccountDetailes* by removing the extension point. Therefore, the *ViewAccountDetailes* is a client in this relationship despite its independence.

The second part of this relationship is the extending use case, the *ViewHistory* use case. It represents another client side of the relationship. *ViewHistory* is not necessarily meaningful by itself, so if specific conditions are met in use case *ViewAccountDetailes*, then *ViewHistory* is performed. The performance of *ViewHistory* is dependent on *ViewAccountDetailes*.

Note: in the extension relationship, the two use cases are clients.

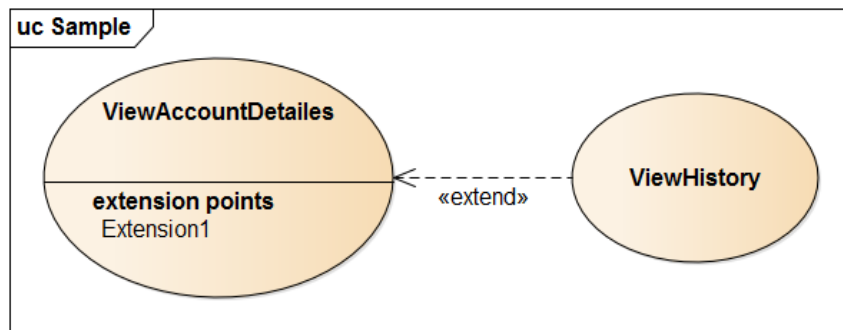
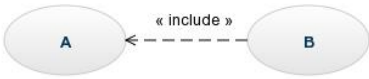
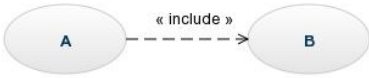
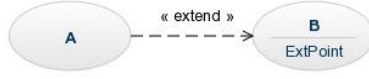




Figure 33 Use Case Extend Relationship Sample

The possible changes that may happen to the include relationship are shown in Table 17. Based on the approach, A and B are clients. A will be affected if something happens to B or to the relationship. B is affected in all cases.

Table 17 Extend Relationship Possible changes

Change Type	The Affected Use Cases	Justification
Change use case A to another one	-	A will be counted as a deleted use case. However, B is not affected.
Change use case B to another one	Use Case A	B will be counted as a deleted use case; therefore A's extension point will be deleted.
Delete use case A	Use Case B	A will be a deleted use case. However, B's existence depends on A.
Delete use case B	Use Case A	A has an extension point because of B, so the deletion of B will have effect on A. A will not have an extension point.
Delete Relationship	Use Case A, Use Case B	B's existence depends on A, so the deletion of the relationship will lead to the deletion of B. Therefore, A will not have an extension point.
	Use Case A, Use Case B	A will be changed to have no extension point. B will depend on A.
	Use Case A, Use Case B	A will be changed to have no extension point. It will depend on B. In addition, B will be a standalone use case.
	Use Case A, Use Case B	B will be changed to have an extension point. A's existence is dependent on B.
	Use Case A, Use Case B	A will be changed to have no extension point. B will depend on A.
	Use Case A, Use Case B	A will be changed to have no extension point. It will depend on B. In addition, B will be a standalone use case.

6.1.8 The Selected UML Use Case Diagram Identifier

The selected identifier here has two main parts, and each part is complete by its own. We have two identifiers. The first identifier is relevant to the use case, which is the use case name. The other identifier is relevant to the actor, which is the actor name. There is separation of the two identifiers because each one identifies a different entity. The possible changes in the identifier are deletion and renaming. Because we cannot detect renaming we will deal with unchanged only.

6.1.9 Summary

The following are the list of all selected elements and attributes, which we are going to evaluate and track their unchanged:

- Actor name.
- Use case name.
- Use case types.
- Actor-actor relationship.
- Actor-use case relationship.
- Generalization relationship.
- Include relationship.
- Extend relationship.

6.2 Terminology and Formalism

This section provides the terminology and formalism of the functional stability metric.

Definition 1 (USE CASE). Let the use case diagram use case be denoted by U . The same use case can have different versions based on different use case diagram versions. Let U_i denote the use case U in use case diagram version i where $i \in [1..n]$.

Definition 2 (ACTOR). Let the use case diagram actor be denoted by A . The same actor can have different versions based on different use case diagram versions. Let A_i denote the actor A in use case diagram version i , where $i \in [1..n]$.

Definition 3 (IDENTIFIER). Let the use case diagram identifier be denoted by ID. The identifier can be either a use case or an actor.

Definition 4 (IDENTIFIER PROPERTIES). Let $P(ID_i)$ denote the set of all properties of the identifier ID in use case diagram version i .

Definition 5 (NUMBER OF USE CASE PROPERTIES). Let NUUP denote the number of all unique properties of the use case U in use case diagram version i and the version we use for comparison.

Definition 6 (USE CASE NAME). Let the use case name in the use case diagram be denoted by UN. The same use case can have only one specific name, which is whatever the use case diagram version is. Otherwise we will consider it as a different use case, because the use case renaming is indefinable.

Definition 7 (USE CASE TYPE). Let the use case type in the use case diagram be denoted by UT.

Definition 8 (ACTOR NAME). Let the actor name in the use case diagram be denoted by AN. The same actor can have only one specific name, which is whatever the use case diagram version is. Otherwise we will consider it as a different actor, because actors renaming is indefinable.

Definition 9 (NUMBER OF USE CASES IN USE CASE DIAGRAM BASE VERSION). Let NU represent the number of use cases in the use case diagram base version.

Definition 10 (NUMBER OF ACTORS IN USE CASE DIAGRAM BASE VERSION). Let NA represent the number of actors in the use case diagram base version.

Definition 11 (NUMBER OF ACTOR RELATIONSHIP). Let NUAR denote the number of all unique relationships of actor A in use case diagram version i and the version we use for comparison.

Definition 12 (IDENTIFIER PROPERTIES CHANGE). Let change that may happen to any identifier be denoted by Ch. Ch represents any change in identifier properties from the use case diagram base version to any other use case diagram version.

Definition 13 (USE CASE UNCHANGED)

UCU is the percentage of unchanged in the use case.

Definition 14 (ACTOR UNCHANGED)

UCA is the percentage of unchanged in the actor.

Definition 15 (FUNCTIONALITY STABILITY)

FS is the percentage of the functional stability, which represents the use case diagram stability.

6.3 Functional Stability Metric

Figure 34 summarizes the computation steps. The measure of the use case diagram stability is done through the following steps:

1. Develop a property change metric for each actor, and each use case. This metric is used to measure the changes of each actor and use cases properties. The unchanged are computed according to Figure 35 and Figure 36, which show use case type changes and use case relationship changes respectively. Figure 37 shows actor relationship changes.

2. Get each use case unchanged, which equals the summation of all use case changes over the number of the use case unique properties (NUUP), Equation 5.1. Dividing by the number of properties will normalize the sum of the use case changes result to be between zero and one. One means all use case diagram use cases have been changed from the i version to the $i+1$ version.
3. Get each actor unchanged, which equals the summation of all actor changes over the number of the actor unique relationships (NUAR), Equation 5.2. Dividing by the number of relationships will normalize the sum of the actor changes result to be between zero and one. One means all use case diagram actors have been changed from the i version to the $i+1$ version.
4. Compute the summation of all use cases and actor change metrics over the summation of NU and NA. Dividing by $NU + NA$ will normalize the summation of use cases and actors change result to be between zero and one. One means all use case diagram elements have been changed from the i version to the $i+1$ version.
5. The overall use case diagram stability metric is computed using Equation 5.3. This final value is also normalized. Zero means all elements have been changed from version i to version $i+1$. Thus, version $i+1$ is unstable. On the other hand, one means nothing has been changed. Therefore, version $i+1$ is completely stable.

To count the changes that may happen to the use cases and actors we have to check first if the use case and actors is still in version $i+1$ or not. If the use case or the actor were deleted, then the change value will be the maximum value, one. The check process is done based on the UN and AN. And then, after confirming the identifier, we will compute each use case and actor change according to Figure 35, Figure 36, and Figure 37.

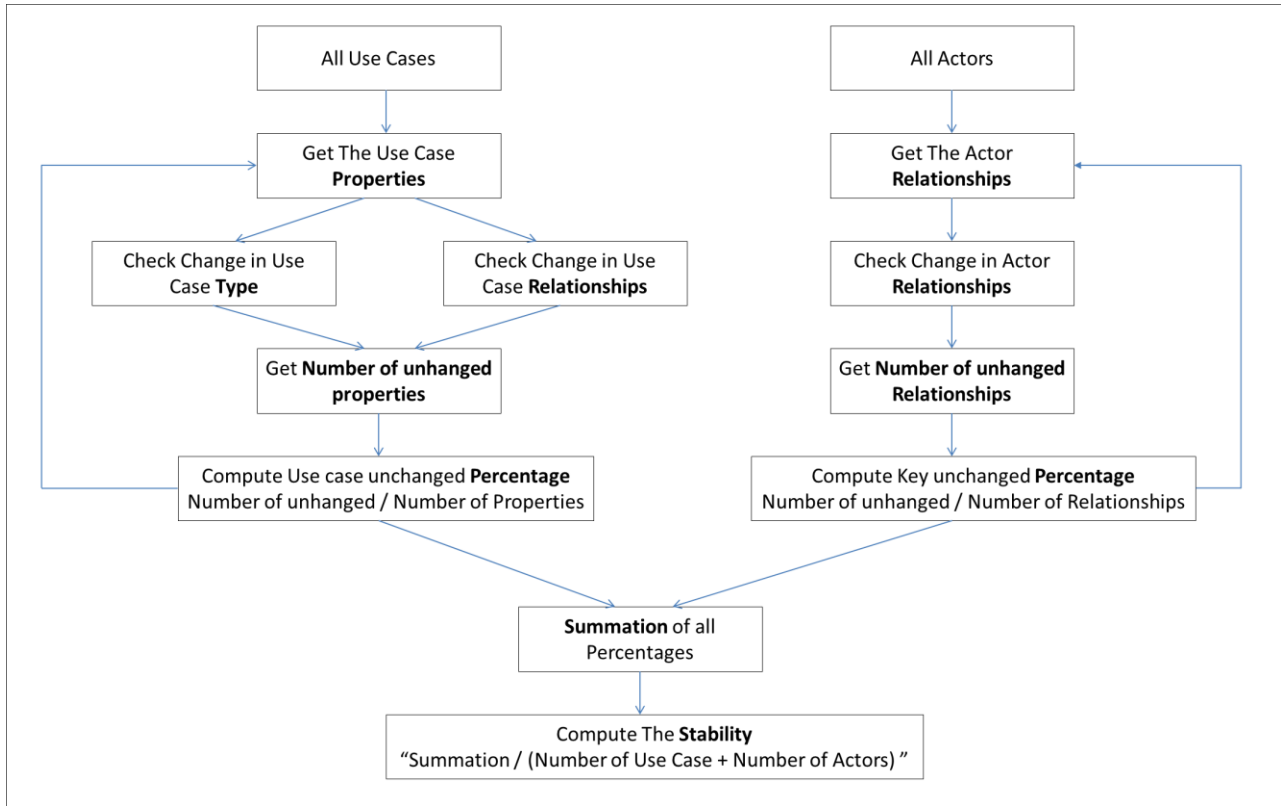


Figure 34 Functional Stability Computation Steps

Figure 35 represents use case type changes. Zero means the use case type remains unchanged. One means that the use case type is changed, either from use case to use case with an extension or from use case with an extension to use case.

$$Ch(UT) = \begin{cases} 0, & \text{Use case Type Changed} \\ 1, & \text{Use case Type Unchanged} \end{cases}$$

Figure 35 Use case Type Changes

Figure 36 represents use case relationships changes. The change counts as one if the relationship is changed. Change will be zero if the relationship remains unchanged.

$$Ch(UR) = \begin{cases} 0, & \text{Relationship Deleted} \\ 0, & \text{Change in Relationship Type} \\ 1, & \text{Relationship Unchanged} \end{cases}$$

Figure 36 Use Case Relationship Changes

Figure 37 represents actor relationships changes. The change counts as one if the relationship is changed. Change will be zero if the relationship remains unchanged.

$$Ch(AR) = \begin{cases} 0, & \text{Relationship Deleted} \\ 0, & \text{Change in Relationship Type} \\ 1, & \text{Relationship Unchanged} \end{cases}$$

Figure 37 Actor Relationships Changes

$$UCU = \frac{Ch(UT(i,i+1)) + \sum_{R=1}^{NUUR} Ch(UR(i,i+1))}{NUUP} \quad 5.1$$

UCU is the abbreviation for Use Case Unchange. This metric computes the unchanged of each use case, which equals the summation of use case type changes and all use case relationship changes over the number of use case properties.

Ch is the abbreviation for Changes in use case types and use case relationships

UR is the abbreviation for Use case Relationship

UT is the abbreviation for Use case Type. NUUP is the abbreviation for Number of Unique Use case Properties.

i : a use case diagram version

$$UCA = \frac{\sum_{R=1}^{NUAR} Ch(AR(i,i+1))}{NUAR} \quad 5.2$$

UCU is the abbreviation for Unchanged in Actor. This metric computes the unchanged of each actor, which equals the summation of actor type changes and all actor relationship changes over the number of actor relationships.

Ch is the abbreviation for Changes in actor relationships

AR is the abbreviation for Actor Relationship

NUAR is the abbreviation for Number of Unique Actor Relationships.

R is the abbreviation for Relationship.

i: a use case diagram version

$$FS(i + 1) = \frac{\sum_{U=1}^{NU} UCU(U) + \sum_{A=1}^{NA} UCA(A)}{NU + NA} \quad 5.3$$

FS is the abbreviation for Functional Stability. This metric computes the stability of the use case diagram, which equals the summation all identifiers' changes subtracted from one.

U is the abbreviation for Use case.

UCU the is abbreviation for Unchanged in Use case.

A is the abbreviation for Actor

UCU is the abbreviation for Unchanged in Actor.

NU is the abbreviation for Number of Use cases in use case diagram version i.

NA is the abbreviation for Number of Actors in use case diagram version i.

i is the abbreviation for a use case diagram version

The following example shows the steps to measure sequence diagram stability.

6.3.1 Example

Figure 38 shows version i of a sample use case diagram, and Figure 39 shows version $i+1$ of the same sample of the use case diagram.

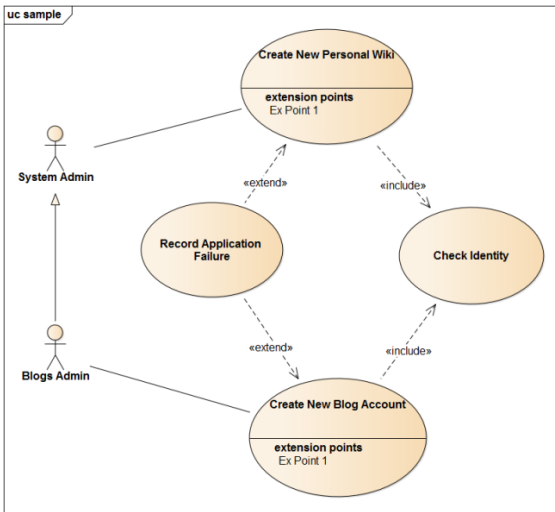


Figure 38 Use Case Sample version i

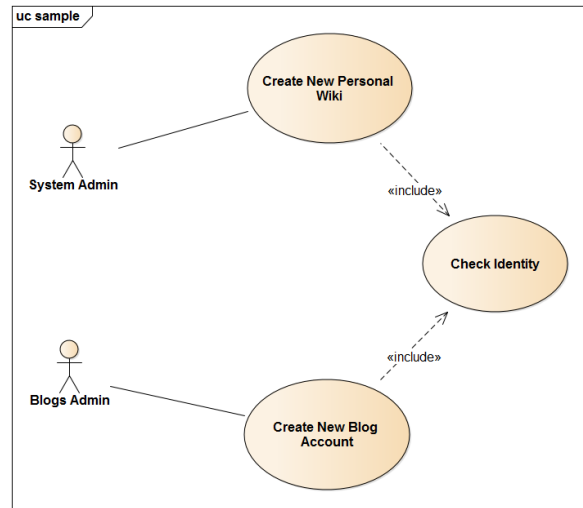


Figure 39 Use Case Sample version $i+1$

Table 18 shows all properties for each version, and Table 19 shows all the changes calculation from sample use case diagram version i to sample use case diagram version $i+1$.

Table 18 Use Case Sample Diagrams Properties

Identifier	Properties	version i Data	version $i+1$ Data
Create New Personal Wiki	Identifier Type	Use case with extension point	Use Case
	Relationships	Check Identity -Include	Check Identity -Include

System Admin	Identifier Type	Actor	Actor
	Relationships	Create New Personal Wiki – Association	Create New Personal Wiki – Association
Record Application Failure	Identifier Type	Use Case	Deleted
	Relationships	Create New Personal Wiki-Extend Create New Blog Account-Extend	
Check Identity	Identifier Type	Use Case	Use Case
	Relationships	-	-
Blog Admin	Identifier Type	Actor	Actor
	Relationships	System Admin – Generalization Create New Blog Account - Association	- Create New Blog Account - Association
Create New Blog Account	Identifier Type	Use case with extension point	Use Case
	Relationships	Check Identity -Include	Check Identity -Include

Table 19 Use Case Sample Changes From version i to version i+1

Identifier	Changes	No. of Unique Properties	Unchanged Value	Unchanged Average
Create New Personal Wiki	Use case with extension point => Use Case	2	1	0.5
System Admin	-	2	2	1
Record Application Failure	IDENTIFIER DELETED	3	0	0
Check Identity	-	1	1	1
Blog Admin	System Admin – Generalization DELETED	3	2	0.66
Create New Blog Account	Use case with extension point => Use Case	2	1	0.5

$$FS(i + 1) = \frac{0.5 + 1 + 0 + 1 + 0.66 + 0.5}{6}$$

$$FS(i + 1) = 0.61$$

The 0.61 means that version $i+1$ of the sample use case diagram's stability is 61%; in other words, version $i+1$ kept 61% of version i 's functionality, elements, and attributes. Sixty-one percent of use cases, actors, and relationships remain in the next version.

CHAPTER 7

BEHAVIORAL STABILITY

This chapter provides the analysis and assessment of the UML sequence diagram and introduces the behavioral stability metric.

7.1 Assessment

A sequence diagram is made up of a collection of participants, lifelines, and messages. Change tracking in the sequence diagram will be based on, and relies on, the messages.

For the sequence diagram, we will apply the client and master approach in a different way. This situation is a bit different from the UML class diagram and UML use case diagram; here we are dealing with message invoking only and we assess each message separately. This point will be clarified in the following sections.

7.1.1 Participant

Participants are the system parts that interact with each other during the sequence and each one has a corresponding lifeline. Participants on a sequence diagram can be named in a number of different ways. Figure 40 shows the general description of the participant.

The first part of the name is the object name, which specifies the name of the instance involved in the interaction. In addition, this part has another attribute-the selector- that identifies which particular instance in a multivalued element is used. The selector is an optional part of the name.

If no object is mentioned in the sequence diagram it means that either no object is required, or that an object without any particular name suffices. An object name and the selector are not always available. This kind of information about the participant may be unspecified during the design process, so we are not going to count any change that may happen to them.

The second part consists of the class name and the decomposition. The class name represents one of the identifier parts. The decomposition is used to point to another interaction diagram that shows details of how this participant processes the message it receives. Thus, it is an optional part, so we are going to ignore it.

From the participant information, only the class name is involved in change assessment. Other parts are neglected, due to the optionality or the absence of its information. We are dealing with the mandatory fields of the participants, such as participant *A* in Figure 41. We can identify the class by its name only, so that if the class name is changed we cannot recognize the original one. Hence, we deal with this case by considering that the class is deleted and a new class has emerged.

```
Object_name [selector] : class_name ref decomposition
```

Figure 40 General Description of Participant

Figure 41 shows a sample of a sequence diagram participant. Classifier *A* calls `message1` from the classifier *B*. In this sample *A* represents the client side of the relation, because it calls `message1` that belongs to *B*, which means that any changes that may happen to this method or to *B* will affect *A*. *B* represents the master side of the relationship, because it does not depend on *A*.

The possible changes here include changing the classifier *A* to another one. This will not affect classifier *B*; as we mentioned earlier, *B* is a master classifier in this relationship. Changing the *B* to another one has an effect on *A*. The type of change on *B* that we are able to recognize here is the deletion of *B*, because classifiers' renaming recognition is not identifiable. The deletion of *B* means that, *message1* is a completely different message than the original one; thus there is a change in $\backslash s$ behavior.

By change participant to another one we mean that it is changing in participant name only, because the only information we have about the participant is its name.

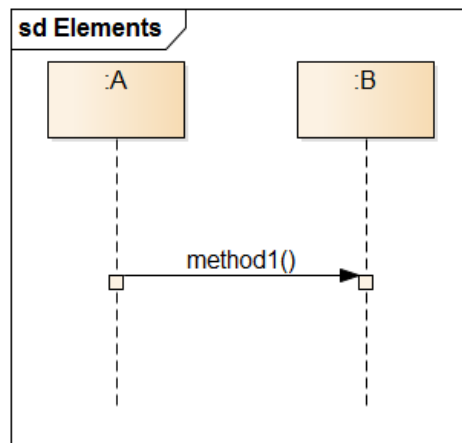


Figure 41 Sequence Diagram Participants

7.1.2 Stereotypes

Stereotypes are used to describe a specific property that a classifier has, which we cannot show in the standard UML classifier. There are three main stereotypes which can be used in sequence diagrams, namely [44, 45]:

- **Entity**: used to represent behavior related to the system data.
- **Boundary**: represents the elements that usually interact with the system actors. Boundaries are called the front-end elements.

- **Controller:** these elements serve as a median between entities and controllers. The controller manages the interaction flow.

We will involve the stereotypes in our assessment, by treating them as if they were usual participants.

7.1.3 Messages

An interaction in a sequence diagram occurs when one participant decides to send a message to another participant, as shown in Figure 41, where *A* sends a message to *B*.

Messages are the heart of the sequence diagram, as they are used to represent the behavior of the systems. Sometimes they are called events, which refer to any point in an interaction where something occurs. We will use the term message because it is the one used by software designers. Messages on a sequence diagram are specified using an arrow from the participant that wants to pass the message (Message Caller), to the participant that receives the message (Message Receiver). Messages can flow in whatever direction makes sense for the required interaction: from left to right, right to left, or even back to the Message Caller itself.

Figure 42 shows the message signature format, which consists of four parts. The attribute, which is used to store the return value of this message, is an optional part. The second part is the message name. The message name is chosen to be the other half of the identifier. The third part of the message signature is the arguments, Figure 43 shows its format. We can specify any number of different arguments on a message, with each separated by a comma. The last part is the return type, which states what the return value from the message will be.

```
attribute = message_name (arguments) : return_type
```

Figure 42 General Description of Message

```
<name>:<class>
```

Figure 43 Message Arguments

The format elements that can be used for a particular message will depend on the information known about a particular message at any given time. For example, message1 in Figure 41, does not indicate that the message clearly has no argument, or return values, but it is the only available information about it. It means that, for now, no further information is known. From that, in order to be consistent and unified with all messages we will involve only the message name information in the stability assessment.

Triggering a message may result in one or more messages being sent by the receiving participant. Those resulting messages are said to be nested within the triggering message, and there can be any number of nested messages and any number of levels on the sequence diagram.

There are five different message types, differentiated based on the message arrow. Each message has its own meaning. For example, the Message Caller may choose to wait for a message to return before carrying on with its work. Or it may choose to just send the message to the Message Receiver without waiting for any return as a form of "fire and forget" message.

7.1.4 A synchronous message

A synchronous message declares that the Message Caller waits for the Message Receiver to return from the message invocation. This can be implemented in the code as a simple method invocation.

Figure 44 shows a sample diagram of a synchronous message, where the figure shows that a classifier *A* is the client, and classifier *B* is the master.

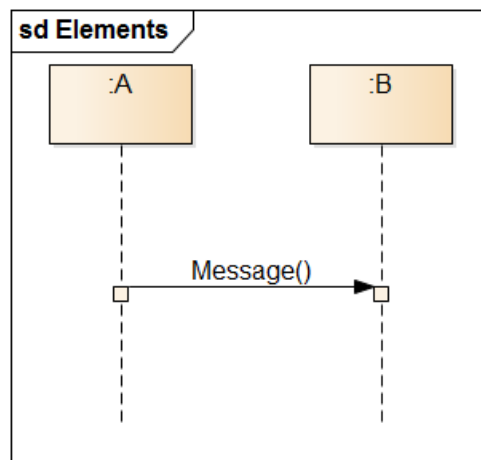


Figure 44 Synchronous Message

Figure 45 explains how we can convert this type of message into a code using a simple method invocation. The *MessageReceiver* which represents participant *B* in our situation, is in the master role of the relationship because it is a standalone participant and does not depend upon participant *A*. The *MessageCaller*, which represents *A* in our situation, is on the client side of the relationship. It depends on *B*. Any change that may happen to *B* will have a direct effect on *A*.

```
public class MessageReceiver{  
    public void foo( ) {  
    }  
}
```

```

public class MessageCaller{

    private MessageReceiver messageReceiver;

    public doSomething(String[] args){
        this.messageReceiver.foo( );
    }
}

```

Figure 45 Implementation Code of Synchronous Message

7.1.5 An asynchronous message

It is not always the case that interactions are happening one after the other. Interactions can happen at the same point in time, and this is what an asynchronous message is about. An asynchronous message declares that Message Caller invokes a message and does not wait for the message invocation to return before carrying on with the rest of the interaction's steps. This means that the Message Caller will invoke a message on the Message Receiver and the Message Caller will be busy invoking further messages before the original message returns, as shown in Figure 46. The figure shows that participant *A* is the client, and participant *B* is the master.

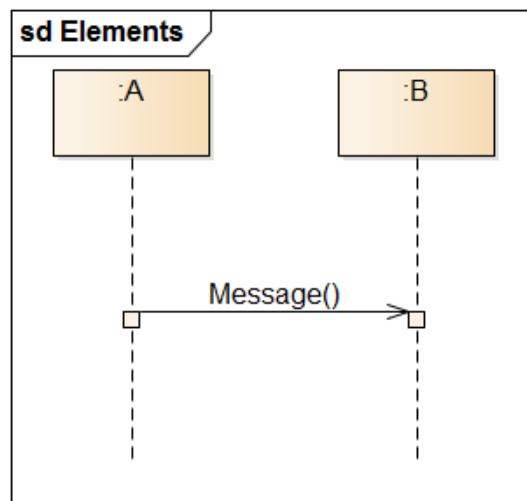


Figure 46 An Asynchronous Message

Figure 47 and Figure 45 explain how we can convert this type of message into a code using threads. The *MessageReceiver*, which represents participant *B* in our situation, is on the master side of the relationship because it is a standalone and does not depend upon *A*. The *MessageCaller*, which represents participant *A* in our situation, is on the client side of the relationship. It depends on *B*. Any change that may happen to *B* will have a direct effect on *A*.

```
public class MessageReceiver implements Runnable {  
    public void operation1( ) {  
        Thread fooWorker = new Thread(this);  
        fooWorker.start();  
    }  
  
    public void run( ) {  
    }  
}  
  
public class MessageCaller  
{  
    private MessageReceiver messageReceiver;  
  
    public void doSomething(String[] args) {  
        this.messageReceiver.operation1( );  
    }  
}
```

Figure 47 Implementation Code of an Asynchronous Message

7.1.6 A return message

The return message, shown in Figure 48, is used at the end of an activation bar. The control flow of the activation is returned to the participant that passed the original message. In code, a return is like reaching the end of the method or calling a return statement. Return messages are an optional notation; their use will make the sequence diagram too busy, so there is no need to show them. However, in synchronous message invocation there is an implied return arrow on the activation

bars that are invoked. We will skip this type of message, and we will not involve it in our assessment.

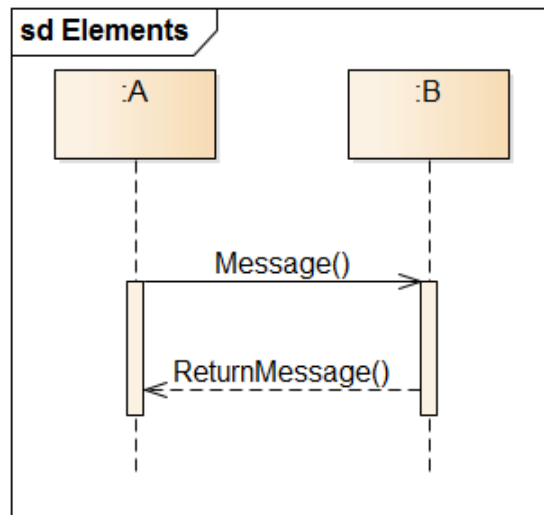


Figure 48 Return Message

7.1.7 Creation Message & Destruction Message

Participants do not necessarily live for the entire duration of a sequence diagram's interaction. Participants can be created and destroyed according to the messages that are being passed, as shown in Figure 49 and Figure 50. A creation message is used to create objects during interactions, while a destruction message is used to delete objects during interactions.

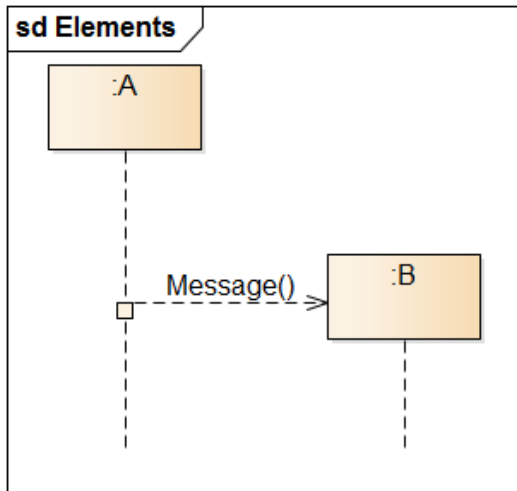


Figure 49 Creation Message

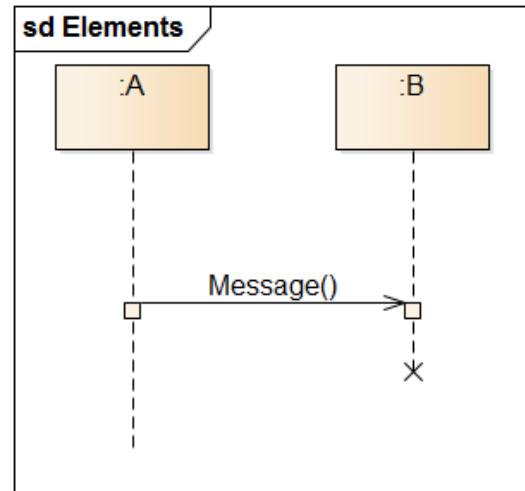


Figure 50 Destruction Message

Figure 51 shows the implementation code of creation message. For the destruction message, we do not always have an explicit destroy method, for example in Java. Showing it on the sequence diagram does not make sense. What happens in Java is that after having finished executing the *doSomething* method, the *MessageReceiver* object will be marked for destruction; after that the garbage collector will implicitly handle the destruction. Thus, in this case there is no need for additional destruction messages.

However, our metric is not focusing only on Java. We are trying to figure out a comprehensive metric that can be applied to whatever the design implementation language may be. In these two types of messages, participant *A* is the master and participant *B* is the client. The *MessageReceiver*, which represents *B*, is on the client side of the relationship because its existence depends upon *A*. The *MessageCaller*, which represents *A*, is on the master side of the relationship because it controls *B*'s existence. Any change that may happen to *A* will have a direct effect on *B*.

```
public class MessageReceiver {
}

public class MessageCaller {

    public void doSomething() {
        MessageReceiver messageReceiver = new MessageReceiver( );
    }
}
```

Figure 51 Implementation Code of Creation Message

7.1.8 Notes, Activation Bars, and Actors

Notes are used to help in associate interactions within elements, place local variable names and values, and place the state invariant information. It is used to describe some information about the diagram. Notes are not used to represent any behavioral states of the diagram. We will skip them.

An activation bar can be shown at the sending and receiving ends of a message. It indicates that the sending participant is active while it sends the message and that the receiving participant is actively doing something after the message has been received. The activation bars are optional, so we will skip them too.

The sequence diagram initiator is a user; a simple label at the top is used rather than a rectangle, and it is the one which initiates the first message. A actor name will be involved in the assessment process.

7.1.9 Time

Sequence diagrams are primarily about the ordering of the interactions between participants. The order that interactions are placed down the diagram indicates the order in which those interactions will take place in time. Time on a sequence diagram is all about ordering, not duration. However,

the time at which an interaction occurs is indicated on a sequence diagram by where it is placed vertically on the diagram. The amount of vertical space the interaction takes up has nothing to do with the duration of time that the interaction will take. We will take the order into our consideration.

7.1.10 The Selected UML Sequence Diagram Identifier

The message represents the identifier in the sequence diagram. In Figure 50, the message is the core of the interaction between participant *A* and participant *B*. The client and master approach that we follow to track changes and avoid assessment duplication will be applied in a different way in the sequence diagram. The messages are the main part of the interactions, so we linked the changes to them directly. First, from the previous evaluation of the participant's relationships, there are no connected participants which are masters and clients at the same time. Each time, one of the participants is a client and the other is a master. So whichever is the client and master, if the message type changes then we have to count the change once. And because we chose the message as the identifier, we will count it for the message that connects the two participants.

A second point in selecting the message as an identifier, is the message order. The sequence diagrams are used to represent the system behavior and to show how the interactions really act. Interactions and order of messages is very important in defining the system's behavior. And, because the order is a message property, we cannot assign the order for the participants. So this is another reason to select the message as the identifier in the sequence diagram. However, in the end, we are tracking the changes of the system behavior, not the changes in the participants

themselves, despite that any change in them reflects on the message properties. So we will focus the messages and their properties' changes only.

The selected identifier is the message name. However, message names cannot be used as an identifier alone. We may have different participants, but with the same message name. So we chose another property beside the name, which is the message receiver. In fact the message receiver here represents the participant which owns the message.

We identified five properties for each message:

- **Message Name:** the invoked message name.
- **Message Receiver:** the participant which owns the method. We consider the message as being a new one if the message receiver is changed, despite having the same name. Some participants may have the same message name, so we cannot specify the original message from its name only.
- **Message Caller:** the participant which initiates the message.
- **Message Type:** we consider four types: synchronous, asynchronous, creation, and destruction. In the case of a return message, we do not consider it because it is an optional message. As we mentioned above, this will represent the client/master changes.
- **Message Order:** we assign a number to every message to indicate its order in the execution process. The base message will have the order number zero.

The changes that may happen to the identifier are shown in Table 20. This table is just to show which parts are affected by a specific change. However, this table will not affect the way we compute the stability. As we mentioned earlier, we connect the changes with the messages.

Table 20 Possible Identifier Changes

Element	Change Type	The Affected Classes
Message Caller	Change the class to another one	One of identifier properties changed
	Deleted	Identifier is deleted
Message Receiver	Change the class to another one	Identifier is deleted
	Deleted	Identifier is deleted
Message Type Asynchronous	Synchronous	Message Receiver
	Creation	Message Caller, Message Receiver
	Destruction	Message Caller, Message Receiver
Message Type Synchronous	Asynchronous	Message Receiver
	Creation	Message Caller, Message Receiver
	Destruction	Message Caller, Message Receiver
Message Type Creation	Asynchronous	Message Caller, Message Receiver
	Synchronous	Message Caller, Message Receiver
	Destruction	Message Caller, Message Receiver
Message Type Destruction	Asynchronous	Message Caller, Message Receiver
	Synchronous	Message Caller, Message Receiver
	Creation	Message Caller, Message Receiver
Message Order	The Order	-

7.2 Terminology and Formalism

This section provides the terminology and formalism of the behavioral stability metric.

Definition 1 (PARTICIPANT). Let the sequence diagram participants be denoted by P . The same participants can have different versions based on different sequence diagram versions. Let P_i denote the participants P in sequence diagram version i , where $i \in [1..n]$.

Definition 2 (MESSAGE PROPERTIES). Let $P(M_i)$ denote the set of all properties of the message M in sequence diagram version i .

Definition 3 (MESSAGE NAME). Let the message name in the sequence diagram be denoted by MN . The same message can have only one specific name, whatever the sequence diagram version is. Otherwise we will consider it to be a different message because the message renaming is indefinable.

Definition 4 (MESSAGE RECEIVER). Let the message receiver in the sequence diagram be denoted by MR . The same message can have only one specific receiver, whatever the sequence diagram version is. Otherwise we will compute it as a different message.

Definition 5 (MESSAGE CALLER). Let the message caller in the sequence diagram be denoted by MC . The same message can have different values based on different sequence diagram versions.

Definition 6 (MESSAGE TYPE). Let the message type in the sequence diagram be denoted by MT . The same message can have different values based on different sequence diagram versions.

Definition 7 (MESSAGE ORDER). Let the message order in the sequence diagram be denoted by MO . The same message can have different values based on different sequence diagram versions.

Definition 8 (NUMBER OF MESSAGES OF SEQUENCE DIAGRAM BASE VERSION).

Let NM represent the number of sequence diagram base version messages.

Definition 9 (MESSAGE PROPERTIES CHANGE). Let the change that may happen to any message be denoted by Ch . Ch represents any change in message properties from the sequence diagram base version to any other sequence diagram version.

Definition 10 (MESSAGE CHANGES)

MC is the percentage of sequence diagram message changes.

Definition 11 (NUMBER OF MESSAGE PROPERTIES). Messages in a sequence diagram have a fixed number of properties, which is three. These properties are: message caller, message type, and message order. We skipped message receiver because we are tracking the changes of the other three. Let the number of message properties be denoted by NMP.

Definition 12 (BEHAVIORAL STABILITY)

BS is the percentage of behavioral stability, which represents the sequence diagram stability.

7.3 Behavioral Stability Metric

We will handle each message property separately and look at the change of the base version. Figure 52 summarizes the computation steps. The measurement of the sequence diagram stability is done through the following steps:

1. Develop a property change metric, which is a metric to measure the unchanged of each message property. Property change is computed according to Figure 54, Figure 55, and Figure 56 and which show message caller changes, message type changes, and message order changes respectively.
2. Get the summation of all property changes metrics and divide it by the number of message properties, Equation 6.1. There are three message properties. Dividing by the number of message properties will normalize the message changes result to be between zero and one. One means all message properties have been fully changed from the i version to the $i+1$ version.

3. Compute the sum of all messages change metrics and divide it by the number of sequence diagram base version messages. Dividing by the number of base version messages will normalize the sum of messages change result to be between zero and one. One means all sequence diagram messages have been changed from the i version to the $i+1$ version.
4. The overall sequence diagram stability metric is computed using Equation 6.2. This final value is also normalized. Zero means all messages have been changed from the version i to the version $i+1$. Thus, the version $i+1$ is unstable. On the other hand, one means nothing has been changed. Therefore, version $i+1$ is completely stable.

To count the changes that may happen to the message properties we have to check first if the message is still in version $i+1$ or not. If the message was deleted, then the message change value will be the maximum value, one. So first we have to check if our message is still in the next version, and this is done based on the name. After that, we have to check the second part of the message identifier, which is the message receiver. Figure 53 shows the possible changes in the message receiver. The message receiver represents one of the message identifier parts. If any change may be happening to it, then we consider the message as another message. So if this property is changed, then the message is fully unstable. After confirming the message we will compute each property change according to Figure 54, Figure 24, and Figure 56.

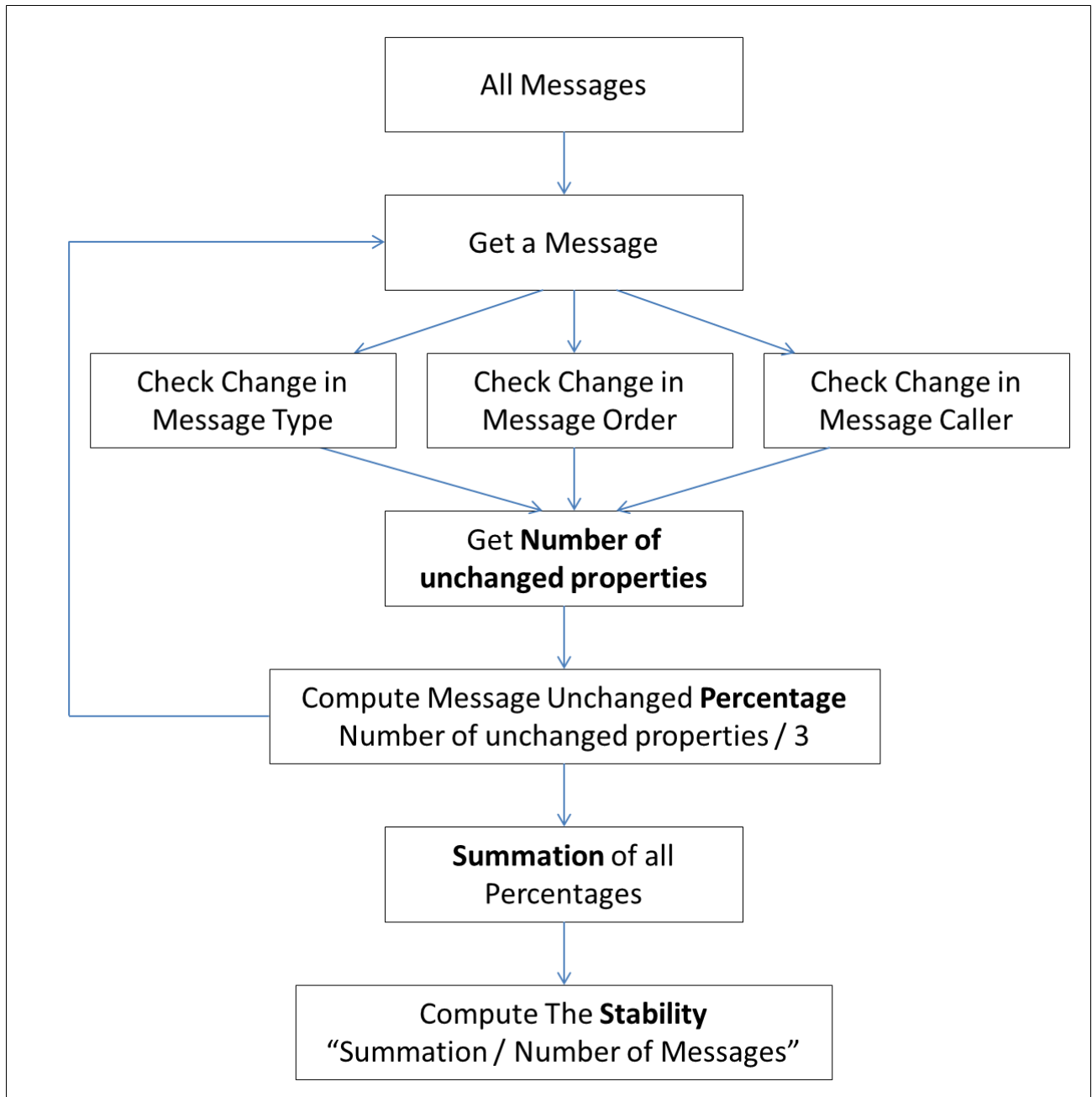


Figure 52 Behavioral Stability Computation Steps

$$Ch(MR) = \begin{cases} 0, & \text{Message Receiver Changed} \\ 1, & \text{Message Receiver Unchanged} \end{cases}$$

Figure 53 Message Receiver Changes

Figure 54 represents message caller changes. Zero means the message caller remains unchanged. One means that the message caller is changed.

$$Ch(MC) = \begin{cases} 0, & \text{Message Caller Changed} \\ 1, & \text{Message Caller Unchanged} \end{cases}$$

Figure 54 Message Caller Changes

Figure 55 represents message type changes. Zero means the message type remains unchanged. One means that the message type is changed. The four possible changes are: asynchronous, synchronous, creation, and destruction.

$$Ch(MT) = \begin{cases} 0, & \text{Message Type Changed} \\ 1, & \text{Message Type Unchanged} \end{cases}$$

Figure 55 Message Type Changes

Figure 56 represents message order changes. Zero means the message order remains unchanged. One means that the message order is changed.

$$Ch(MO) = \begin{cases} 0, & \text{Message Order Changed} \\ 1, & \text{Message Order Unchanged} \end{cases}$$

Figure 56 Message Order Changes

$$UCM = \frac{Ch(MC(i,i+1)) + Ch(MT(i,i+1)) + Ch(MO(i,i+1))}{NMP} \quad 6.1$$

UCM is the abbreviation for Unchanged in Messages. This metric computes the unchanged of each message, which equals the summation of the message caller changes, message type changes, and message order changes over the number of message properties.

UCM is the abbreviation for Unchanged in Messages.

Ch is the abbreviation for Changes in message caller, type, and order.

MC is the abbreviation for Message Caller.

MT is the abbreviation for Message Type.

MO is the abbreviation for Message Order.

NMP is the abbreviation for Number of Message Properties (three); message properties are classified by the caller, type, and order.

i is the abbreviation for a sequence diagram version.

$$BS(i + 1) = \frac{\sum_{M=1}^{NM} UCM(M)}{NM} \quad 6.2$$

BS is the abbreviation for Behavioral Stability. This metric computes the stability of the sequence diagram, which equals the summation all messages changes subtracted from one.

UCM is the abbreviation for Unchanged in Messages.

M is the abbreviation for Message.

NM is the abbreviation for Number of Messages in the sequence diagram version i.

i is the abbreviation for a sequence diagram version.

The following example shows the steps to measure sequence diagram stability.

7.3.1 Example

Figure 57 shows version i of a sample sequence diagram, and Figure 58 shows version $i+1$ of the same sample of a sequence diagram.

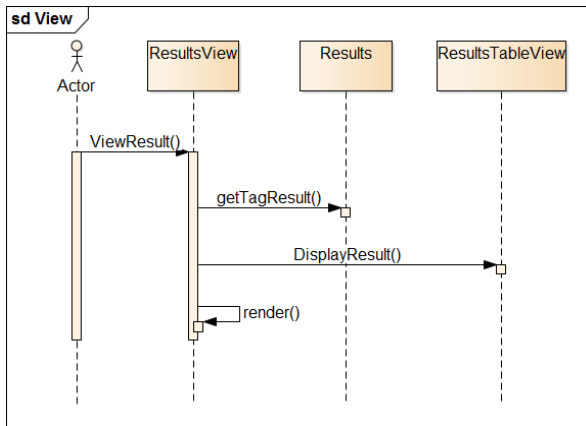


Figure 57 Sequence Diagram Sample version i

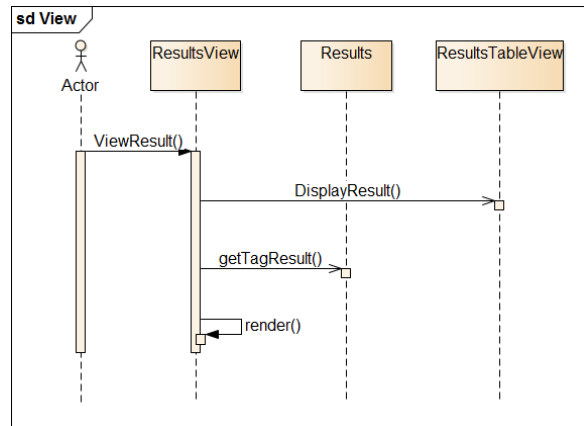


Figure 58 Sequence Diagram Sample version $i+1$

Table 21 shows all message properties for each version. Table 22 shows all the changes calculation from sample sequence diagram version i to sample sequence diagram version $i+1$.

Table 21 All Messages Property for the Sequence Sample Diagrams

Identifier	Properties	version i Data	version $i+1$ Data
viewResult	Message Receiver	ResultsView	ResultsView
	Message Caller	Actor	Actor
	Message Type	Synchronous	Synchronous
	Message Order	1	1
getTagResult	Message Receiver	Results	Results
	Message Caller	ResultsView	ResultsView
	Message Type	Synchronous	Asynchronous
	Message Order	2	3
displayResult	Message Receiver	ResultsTableView	ResultsTableView

	Message Caller	ResultsView	ResultsView
	Message Type	Synchronous	Asynchronous
	Message Order	3	2
render	Message Receiver	ResultsView	ResultsView
	Message Caller	ResultsView	ResultsView
	Message Type	Synchronous	Synchronous
	Message Order	4	4

Table 22 Changes From version 1 to version 2 in Sequence Sample Diagrams

Identifier	Changes	No. of Properties	Unchanged Value	Unchanged Average
viewResult	-	3	3	1
getTagResult	Synchronous => Asynchronous 2 => 3	3	1	0.33
displayResult	Synchronous => Asynchronous 3 => 2	3	1	0.33
render	-	3	3	1

$$BS(i + 1) = \frac{1 + 0.33 + 0.33 + 1}{4}$$

$$BS(i + 1) = 0.66$$

The 0.66 means that version *i+1* of the sample sequence diagram’s stability is 66%. In other words, version *i+1* kept 66% of version *i*’s behavior, elements, and attributes. Sixty-six percent of participants, messages, and relationships remain in the next version.

CHAPTER 8

THEORETICAL VALIDATION

In this chapter, we are going to validate our metrics suite theoretically.

Our proposed metrics aim to capture information about the system stability. Introducing any kind of measurements needs a proper validation, so it must have a scientific basis [46]. Therefore the metrics validation, whether theoretical or empirical, is not a purely objective exercise [47]. The basic question whenever you propose a metric is whether the measure captures the attribute it claims to depict. Accepting a product measure is the process of guaranteeing that the measure of the claimed attribute is a fitting numerical characterization by demonstrating that the representation condition is satisfied [48]. In other words, the theoretical validation confirms that the measure does not abuse any essential properties of the measurement elements [49].

Several frameworks are proposed to validate software metrics. Briand et al [50] proposed a framework to validate cohesion metrics. Weyuker [51] introduced a framework to validate complexity metrics. However, no frameworks were found specifically to validate stability metrics. Therefore we used a standard metrics validation frameworks, Kitchenham's framework, in order to validate our metrics theoretically.

Kitchenham et al. introduced the metric-evaluation framework [52] to validate software metrics. They define various properties that a theoretically valid software metric should have. They identified a set of theoretical criteria that must be satisfied in order to propose a valid measure. The metric-evaluation framework consists of five models: unit definition model, attribute

relationship model, instrumentation model, measurement protocol model, and entity population model.

For a **Unit definition model**, a unit is defined for all measures, including ratio, scale, nominal, and ordinal. There are four types of the unit definition model: reference to a wider theory model, reference to a standard model, reference to a model involving several attributes model, and reference to conversion from another unit model. A metric has a valid unit if the used units are an appropriate means of measuring the attribute.

Reference to a standard determines a metric unit based on an application domain standard. Reference to a wider theory defines the unit for a metric based on the way in which an attribute is observed in a particular entity. Reference to conversion from another unit sets a metric unit by converting from a known unit.

Reference to a model defines the unit of a composite metric by combining the units of the individual metrics involved.

The **Instrumentation model** defines the method used to perform the measurements. The instrumentation model is closely related to the unit definition. It has two types: the direct representational model and the indirect theory-based model. A metric has a valid instrument if the underlying measurement instrument is valid and adjusted properly.

An attribute is may be composed of other attributes, so the **attribute relationship model** is used to define the relationships among these attributes. There are two types of attribute relationship models, namely the definition model and predictive model.

The definition model is used to define a multi-dimensional attribute, while, the predictive model is used in the prediction of a specific attribute value based on other values.

The Measurement protocol model is concerned with how to measure an attribute consistently on a particular entity. The measurement protocol model's aim is to make a measure independent of the environment and the measurer. A metric has a valid protocol if a widely accepted measurement protocol is used.

The Entity population model sets the normal values of a metric.

Kitchenham et al. introduced four properties which every metric must satisfy in order to be theoretically valid. These properties are:

1. "For an attribute to be measurable, it must allow different entities to be distinguished from one another". This means, there must be a two entities with different measurement values.
2. "A valid measure must obey the Representation Condition". For example, in our case, if we have two entities and the first entity is less than the other entity in terms of selected properties, then the stability of the first entity must be less than the second one.
3. "Each unit of an attribute contributing to a valid measure is equivalent". This means that the entities that are measured alongside each other are equivalent [53].
4. "Different entities can have the same attribute value (within the limits of measurement error)".

8.1 Structural Stability Metric Validation

Our proposed metric for measuring class diagram stability have the following parameters:

- The entity is the class diagram being analyzed.
- The attribute measured is the stability.
- The unit is the percentage.
- The data scale is an interval.

The SS (Structural Stability) conforms to Kitchenham's properties as follows:

Property 1:

Suppose we have two versions of a system, version i and version j where $(j > i)$. Assume we have two classes, a class $C1i$ in release i , and a corresponding class version $C1j$ in release j . Let us assume another two classes, a class $C2i$ in release i , and a corresponding class version $C2j$ in release j . Suppose the $C1i$ class has P1 properties, which are a_1, a_2, \dots, a_n and b_1, b_2, \dots, b_x ($x \leq n$), where the set of properties has remained unchanged between the two releases, release i and release j . As well, suppose the $C2i$ class has P1 properties, which are c_1, c_2, \dots, c_n and d_1, d_2, \dots, d_y ($y \leq n$) a set of properties has remained unchanged between the two releases, release i and release j . When $x / P1 \neq y / P1$, then $\text{Stability}(C1) \neq \text{Stability}(C2)$.

Property 2:

Suppose we have two versions of a system, version i and version j where $(j > i)$. Assume we have two classes, a class $C1i$ in release i , and a corresponding class version $C1j$ in release j . Let us assume another two classes, a class $C2i$ in release i , and a corresponding class version $C2j$ in

release j . Suppose the $C1i$ class has $P1$ properties, which are $a1, a2, \dots, an$ and $b1, b2, \dots, bx$ ($x \leq n$) a set of properties has remained unchanged between the two releases, release i and release j . As well, suppose the $C2i$ class has $P1$ properties, which are $c1, c2, \dots, cn$ and $d1, d2, \dots, dy$ ($y \leq n$) a set of properties has remained unchanged between the two releases, release i and release j . When $x / P1 > y / P1$, then $\text{Stability}(C1) > \text{Stability}(C2)$.

Property 3:

Suppose we have two versions of a system, version i and version j where ($j > i$). Assume we have two classes, a class $C1i$ in release i , and a corresponding class version $C1j$ in release j . Let us assume another two classes, a class $C2i$ in release i , and a corresponding class version $C2j$ in release j . Suppose the $C1i$ class has $P1$ properties, which are $a1, a2, \dots, an$ and $b1, b2, \dots, bx$ ($x < n$) a set of properties has remained unchanged between the two releases, release i and release j . As well, suppose the $C2i$ class has $P1$ properties, which are $a1, a2, \dots, an$ and $b1, b2, \dots, b(x+1)$ ($x < n$) a set of properties has remained unchanged between the two releases, release i and release j . Then $\text{Stability}(C2) = \text{Stability}(C1) + 1/P1$.

Property 4:

Suppose we have two versions of a system, version i and version j where ($j > i$). Assume we have two classes, a class $C1i$ in release i , and a corresponding class version $C1j$ in release j . Let us assume another two classes, a class $C2i$ in release i , and a corresponding class version $C2j$ in release j . Suppose the $C1i$ class has $P1$ properties, which are $a1, a2, \dots, an$ and $b1, b2, \dots, bx$ (x

$\leq n$) a set of properties has remained unchanged between the two releases, release i and release j . As well, suppose the $C2i$ class has $P1$ properties, which are a_1, a_2, \dots, a_n and b_1, b_2, \dots, b_x ($x \leq n$) a set of properties has remained unchanged between the two releases, release i and release j . Then $\text{Stability}(C1) = \text{Stability}(C2)$.

8.2 Functional Stability Metric Validation

Our proposed metric for measuring use case diagram stability has the following parameters:

- The entity is the use case diagram being analyzed.
- The attribute measured is the stability.
- The unit is the percentage.
- The data scale is an interval.

The FS (Functional Stability) conforms to Kitchenham's properties as follows:

Property 1:

Suppose we have two versions of a system, version i and version j where ($j > i$). Assume we have two identifiers, an identifier $ID1i$ in release i , and a corresponding identifier version $ID1j$ in release j . Let us assume another two identifiers, an identifier $ID2i$ in release i , and a corresponding identifier version $ID2j$ in release j . Suppose the identifier $ID1i$ has $P1$ properties, which are a_1, a_2, \dots, a_n and b_1, b_2, \dots, b_x ($x \leq n$) a set of properties has remained unchanged between the two releases, release i and release j . As well, suppose the identifier $ID2i$ has $P1$ properties, which are c_1, c_2, \dots, c_n and d_1, d_2, \dots, d_y ($y \leq n$) a set of properties has remained unchanged between the two releases, release i and release j . When $x / P1 \neq y / P1$, then $\text{Stability}(ID1) \neq \text{Stability}(ID2)$.

Property 2:

Suppose we have two versions of a system, version i and version j where $(j > i)$. Assume we have two identifiers, an identifier $ID1i$ in release i , and a corresponding identifier version $ID1j$ in release j . Let us assume another two identifiers, an identifier $ID2i$ in release i , and a corresponding identifier version $ID2j$ in release j . Suppose the identifier $ID1i$ has $P1$ properties, which are $a1, a2, \dots, an$ and $b1, b2, \dots, bx$ ($x \leq n$) a set of properties has remained unchanged between the two releases, release i and release j . As well, suppose the $ID2i$ class has $P1$ properties, which are $c1, c2, \dots, cn$ and $d1, d2, \dots, dy$ ($y \leq n$) a set of properties has remained unchanged between the two releases, release i and release j . When $x / P1 > y / P1$, then $\text{Stability}(ID1) > \text{Stability}(ID2)$.

Property 3:

Suppose we have two versions of a system, version i and version j where $(j > i)$. Assume we have two identifiers, an identifier $ID1i$ in release i , and a corresponding identifier version $ID1j$ in release j . Let us assume another two identifiers, an identifier $ID2i$ in release i , and a corresponding identifier version $ID2j$ in release j . Suppose the identifier $ID1i$ has $P1$ properties, which are $a1, a2, \dots, an$ and $b1, b2, \dots, bx$ ($x < n$) a set of properties has remained unchanged between the two releases, release i and release j . As well, suppose the identifier $ID2i$ has $P1$ properties, which are $a1, a2, \dots, an$ and $b1, b2, \dots, b(x+1)$ ($x < n$) a set of properties has remained unchanged between the two releases, release i and release j . Then $\text{Stability}(ID2) = \text{Stability}(ID1) + 1/P1$.

Property 4:

Suppose we have two versions of a system, version i and version j where $(j > i)$. Assume we have two identifiers, an identifier $ID1i$ in release i , and a corresponding identifier version $ID1j$ in release j . Let us assume another two identifiers, a identifier $ID2i$ in release i , and a corresponding identifier version $ID2j$ in release j . Suppose the identifier $ID1i$ has $P1$ properties, which are $a1, a2, \dots, an$ and $b1, b2, \dots, bx$ ($x \leq n$) a set of properties has remained unchanged between the two releases, release i and release j . As well, suppose the identifier $ID2i$ has $P1$ properties, which are $a1, a2, \dots, an$ and $b1, b2, \dots, bx$ ($x \leq n$) a set of properties has remained unchanged between the two releases, release i and release j . Then $\text{Stability}(ID1) = \text{Stability}(ID2)$.

8.3 Behavioral Stability Metric Validation

Our proposed metric for measuring sequence diagram stability has the following parameters:

- The entity is the sequence diagram being analyzed.
- The attribute measured is the stability.
- The unit is the percentage.
- The data scale is an interval.

The BS (Behavioral Stability) conforms to Kitchenham's properties as follows:

Property 1:

Suppose we have two versions of a system, version i and version j where $(j > i)$. Assume we have two messages, a message $M1i$ in release i , and a corresponding message version $M1j$ in release j . Let us assume another two messages, a message $M2i$ in release i , and a corresponding message

version $M2j$ in release j . Suppose the $M1i$ class has the properties $a1, a2, a3$ and $M1j$ has the properties $b1, b2, b3$, with x ($x \leq 3$) and properties have remained unchanged between the two releases, release i and release j . As well, suppose the $M2i$ class has the properties $c1, c2, c3$ and $M2j$ has the properties $d1, d2, d3$ with y ($y \leq 3$), and properties have remained unchanged between the two releases, release i and release j . When $x / 3 \neq y / 3$, then $\text{Stability}(M1) \neq \text{Stability}(M2)$.

Property 2:

Suppose we have two versions of a system, version i and version j where ($j > i$). Assume we have two messages, a message $M1i$ in release i , and a corresponding message version $M1j$ in release j . Let us assume another two messages, a message $M2i$ in release i , and a corresponding message version $M2j$ in release j . Suppose the $M1i$ class has the properties $a1, a2, a3$ and $M1j$ has the properties $b1, b2, b3$, with x ($x \leq 3$) properties have remained unchanged between the two releases, release i and release j . As well, suppose the $M2i$ class has the properties $c1, c2, c3$ and $M2j$ has the properties $d1, d2, d3$ with y ($y \leq 3$) properties have remained unchanged between the two releases, release i and release j . When $x / 3 > y / 3$, then $\text{Stability}(M1) > \text{Stability}(M2)$.

Property 3:

Suppose we have two versions of a system, version i and version j where ($j > i$). Assume we have two messages, a message $M1i$ in release i , and a corresponding message version $M1j$ in release j . Let us assume another two messages, a message $M2i$ in release i , and a corresponding message version $M2j$ in release j . Suppose the $M1i$ class has the properties $a1, a2, a3$ and $M1j$ has the

properties b_1, b_2, b_3 , with x ($x < 3$), and properties have remained unchanged between the two releases, release i and release j . As well, suppose the $M2i$ class has the properties a_1, a_2, a_3 and $M2j$ has the properties b_1, b_2, b_3 with $x+1$ and properties have remained unchanged between the two releases, release i and release j . When $x/3 > y/3$, then $\text{Stability}(M2) = \text{Stability}(M1) + 1/3$.

Property 4:

Suppose we have two versions of a system, version i and version j where ($j > i$). Assume we have two messages, a message $M1i$ in release i , and a corresponding message version $M1j$ in release j . Let us assume another two messages, a message $M2i$ in release i , and a corresponding message version $M2j$ in release j . Suppose the $M1i$ class has the properties a_1, a_2, a_3 and $M1j$ has the properties b_1, b_2, b_3 , with x ($x \leq 3$) properties have remained unchanged between the two releases, release i and release j . As well, suppose the $M2i$ class has the properties a_1, a_2, a_3 and $M2j$ has the properties b_1, b_2, b_3 with x ($x \leq 3$) properties have remained unchanged between the two releases, release i and release j . Then $\text{Stability}(M1) = \text{Stability}(M2)$.

CHAPTER 9

CASE STUDIES

In this chapter, we describe the case studies.

We selected case studies in our experiment from two groups. The first group is published case studies, and we have selected three different case studies. The other group consists of student projects. These projects were designed by undergraduate students as a senior project conducted at King Fahd University of Petroleum and Minerals, and another three projects were selected from the best of these.

Before starting the experiment, we created a second version from each UML diagram. Our creation of the diagram takes into consideration the most likely changes that can be introduced without affecting the core of the original one. Next, we perform our experiment manually because we do not have a tool that helps in conducting our experiment. The next sections show all the case studies in detail.

9.1 Case Study 1: ATM

Automated Teller Machine (ATM) is a well-known case study [54]. The customer inserts his card, enters a PIN and then can perform transactions, such as withdrawal and deposit, before a receipt is issued by the ATM at the end of all the transactions. We used its class diagram and sequence diagrams. For the sequence diagram we selected three diagrams, and the average number of messages is seven. Table 23 shows the ATM experimental summary.

Figure 59 and Figure 60 shows ATM class diagram version 1 and version 2 respectively. Table 24 displays the comparison, and Table 25 shows the computation results. Sixty-nine percent of version 1 of the class diagram remains in version 2.

Figure 61 and Figure 62 show the first version and second version of the ATMStartUp sequence diagram respectively. The comparison is described in Table 26. The second version of the sequence diagram kept 76% of the first version, as shown in the Table 27 computations.

Figure 63 and Figure 64 show the Deposit sequence diagram, version 1 and version 2, respectively. The comparison is shown in Table 28. Eighty-three percent of the first version remains in the second one, as shown in the Table 29 computations.

Figure 65 and Figure 66 shows the first version and second version of the Withdrawal sequence diagram respectively. The comparison is described at Table 30. The second version of the sequence diagram kept 62% of the first one as, shown in the Table 31 computations.

Table 23 ATM Case Study Summary

Diagram Type	System Name	Stability
Class Diagram	ATM	0.694
Sequence Diagram	ATMStartUp	0.761
Sequence Diagram	Deposit	0.833
Sequence Diagram	Withdrawal	0.62

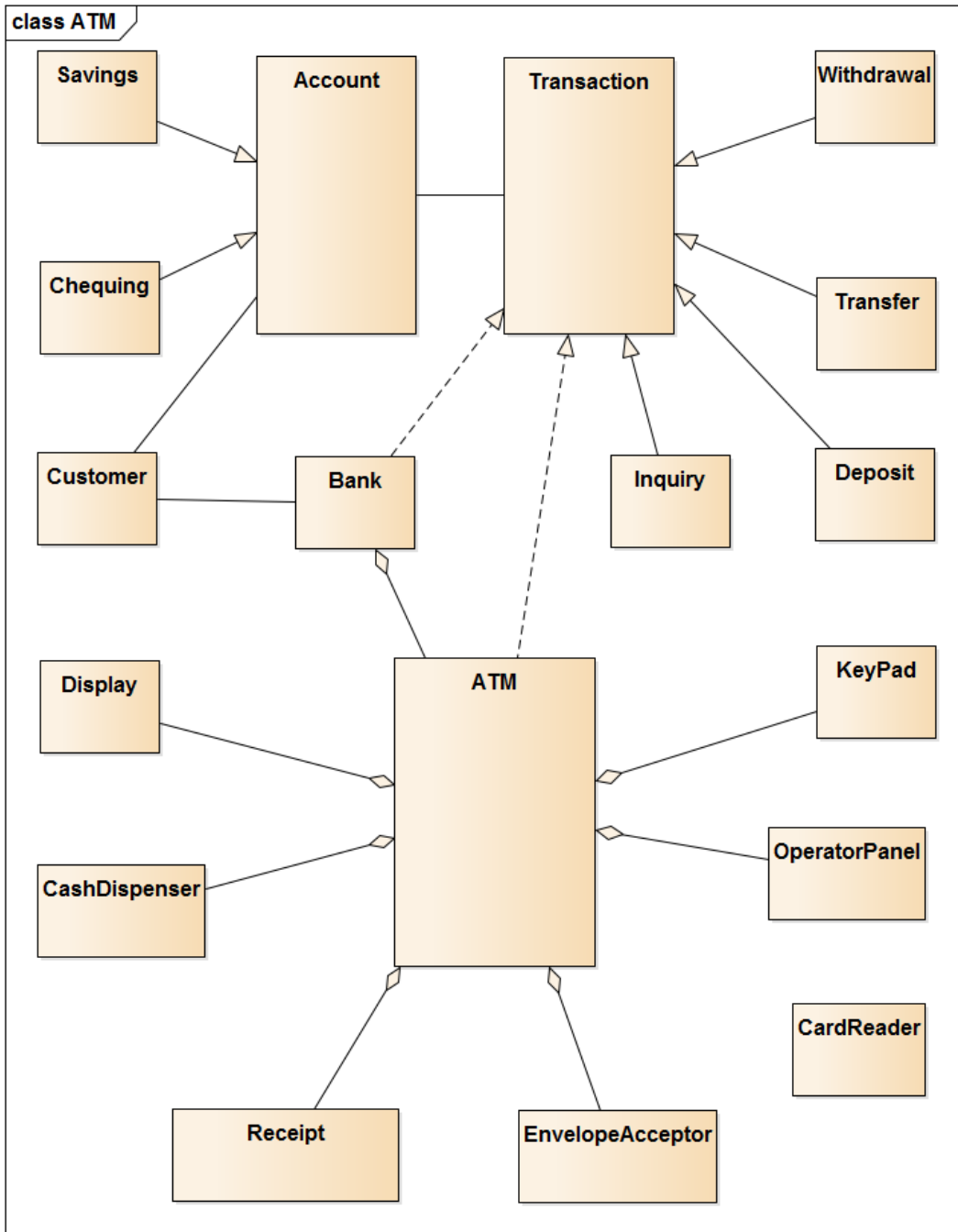


Figure 59 ATM Class Diagram version 1

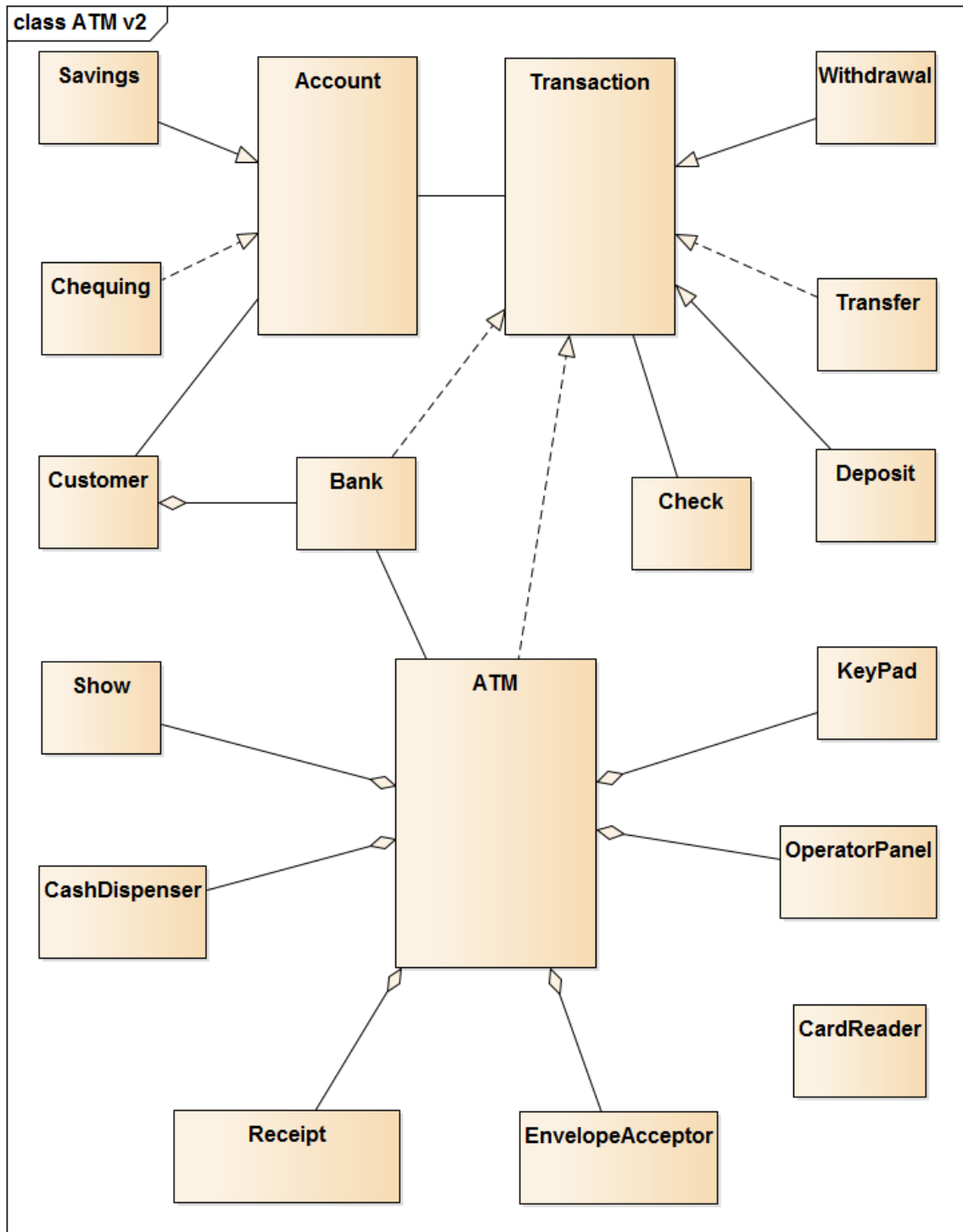


Figure 60 ATM Class Diagram v

Table 24 ATM Class Diagrams Comparison

Classifier Name	Version 1		Version 2	
	Classifier Type	Classifier Relationships	Classifier Type	Classifier Relationships
Savings	Class	Account-INH	Class	Account-INH
Account	Class	Customer-ASO	Interface	-
		Transaction-ASO		Transaction-ASO
Transaction	Interface	Account-ASO	Interface	Account-ASO
				Check-ASO
Withdrawal	Class	Transaction-INH	Class	Transaction-INH
Chequing	Class	Account-INH	Class	Account-REA
Transfer	Class	Transaction-INH	Class	Transaction-REA
Customer	Class	Account-ASO	Class	Account-INH
		Bank-ASO		Bank-AGG
Bank	Class	Customer-ASO	Class	-
		Transaction-REA		Transaction-REA
		ATM-AGG		ATM-ASO
Inquiry	Class	Transaction-INH	DELETED	
Deposit	Class	Transaction-INH	Class	Transaction-INH

Display	Class	-	DELETED	
ATM	Class	Display-AGG	Class	Show-AGG
		CashDispenser-AGG		CashDispenser-AGG
		Receipt-AGG		Receipt-AGG
		EnvelopeAcceptor-AGG		EnvelopeAcceptor-AGG
		CardReader-AGG		CardReader-AGG
		OperatorPanel-AGG		OperatorPanel-COM
		KeyPad-AGG		KeyPad-AGG
KeyPad	Class	-	Class	-
CashDispenser	Class	-	Class	-
OperatorPanel	Class	-	Class	-
Receipt	Class	-	Class	-
EnvelopeAcceptor	Class	-	Class	-
CardReader	Class	-	Class	-

Table 25 ATM Class Diagram Comparison Results

Classifier Name	Changes From version 1 to version 2	Number Of Changes	Number Of Unique Pairs	Changes / Unique
Savings	-	0	2	0
Account	Class => Interface	2	3	0.666
	Customer-ASO => DELETED			
Transaction	Check-ASO => NEW	1	3	0.333
Withdrawal	-	0	2	0
Chequing	Account-INH => Account-REA	1	2	0.5
Transfer	Transaction-INH => Transaction-REA	1	2	0.5
Customer	Account-ASO => Account-INH	2	3	0.666
	Bank-ASO => Bank-AGG			
Bank	Customer-ASO => DELETED	2	4	0.5
	ATM-AGG => ATM-ASO			
Inquiry	DELETED	FULL	FULL	1
Deposit	-	0	2	0
Display	DELETED	FULL	FULL	1
ATM	Display-AGG => Deleted	3	9	0.333
	Show-AGG => NEW			
	OperatorPanel-AGG => OperatorPanel-COM			
KeyPad	-	0	1	0
CashDispenser	-	0	1	0

OperatorPanel	-	0	1	0
Receipt	-	0	1	0
EnvelopeAcceptor	-	0	1	0
CardReader	-	0	1	0
SUM				5.5
Instability				0.305
Stability				0.695

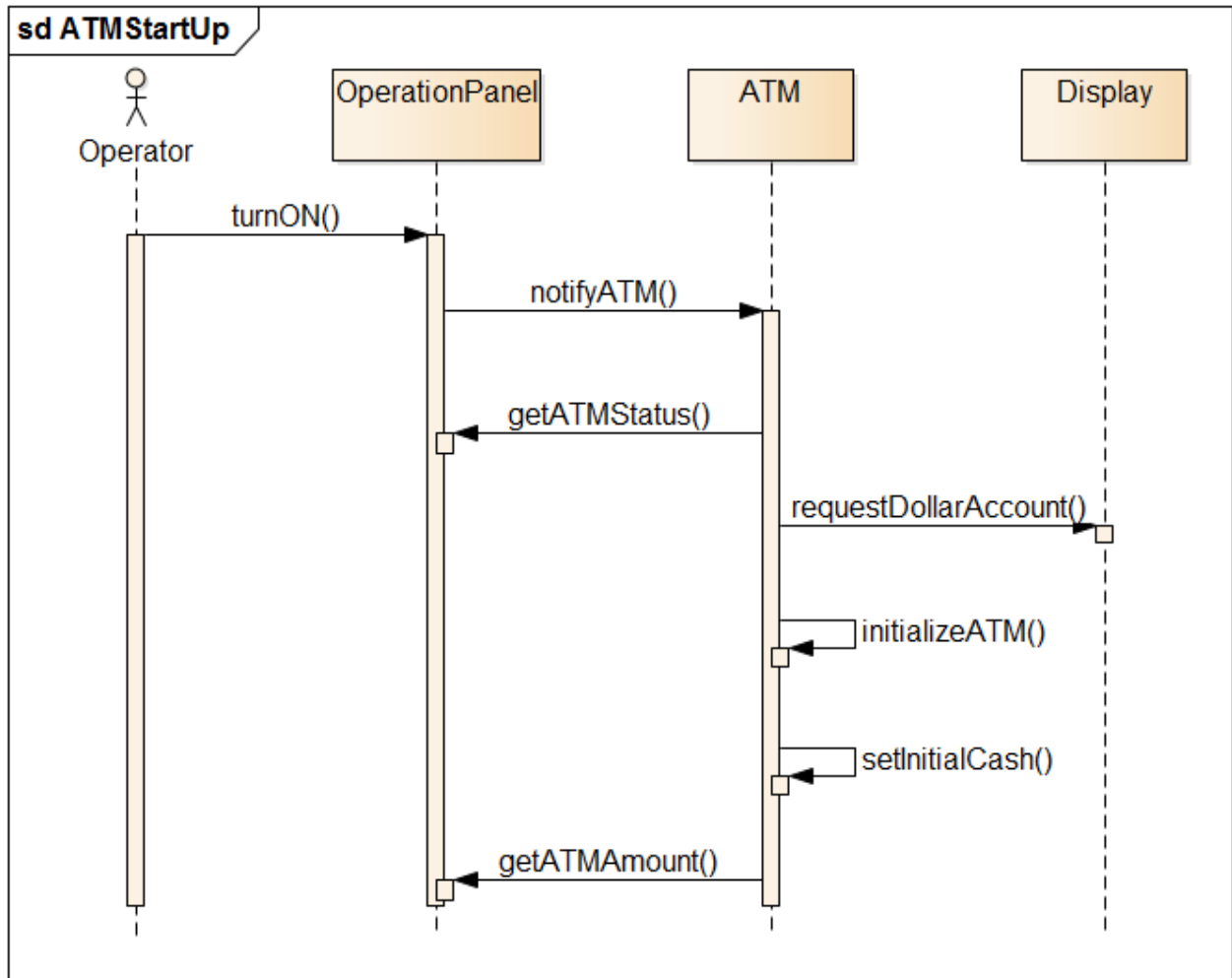


Figure 61 ATMStartUp Sequence Diagram version 2

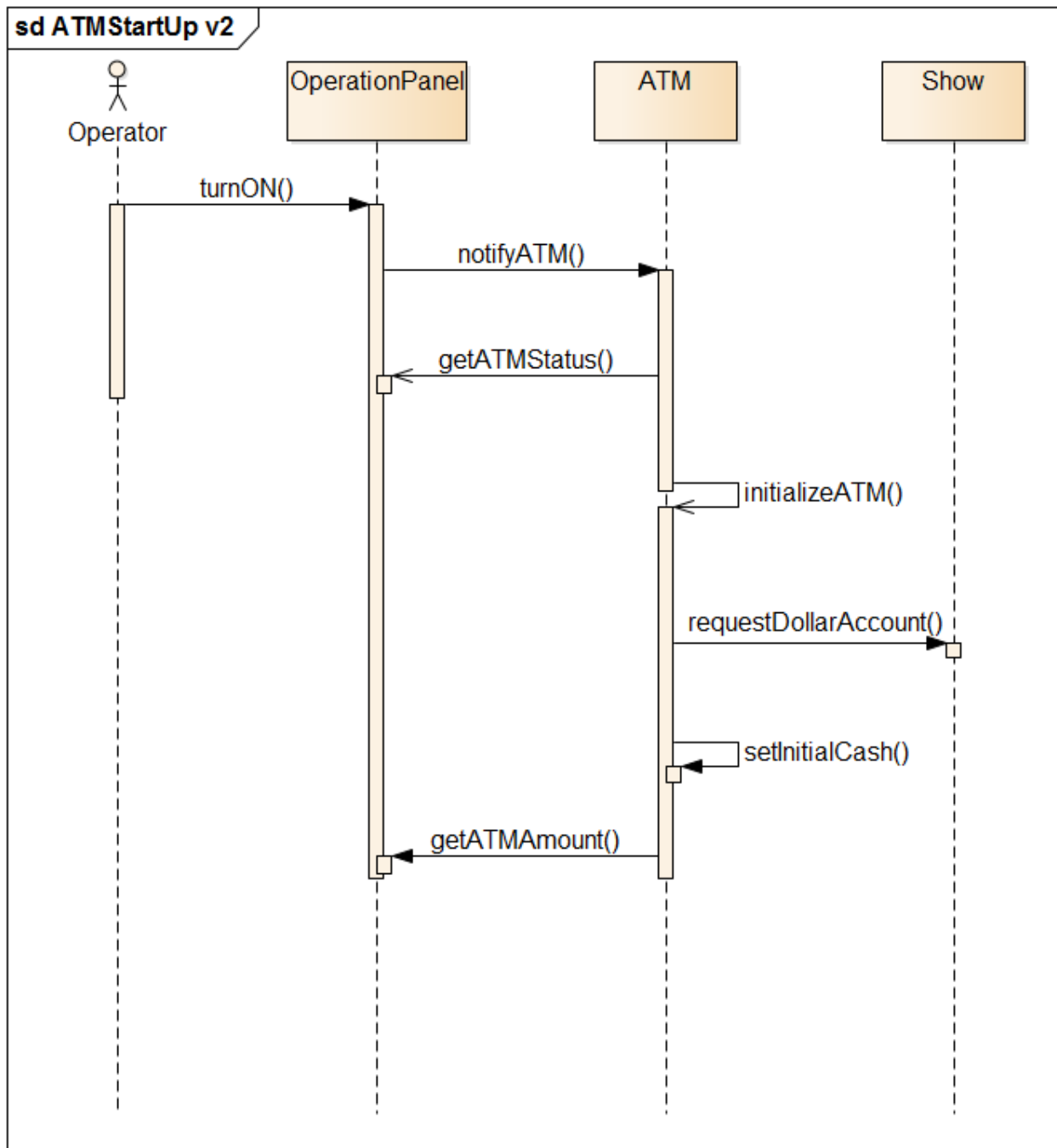


Figure 62 ATMStartUp Sequence Diagram version 2

Table 26 ATMStartUp Sequence Diagrams Comparison

The Identifier		Properties					
Message Name	Message Receiver	Message Caller		Message Type		Message Order	
		version 1	version 2	version 1	version 2	version 1	version 2
teurnON	OperationPanel	Operator	Operator	Synchronous	Synchronous	1	1
notifyATM	ATM	OperationPanel	OperationPanel	Synchronous	Synchronous	2	2
getATMStatus	OperationPanel	ATM	ATM	Synchronous	Asynchronous	3	3
requestDollarAccount	Display	DELETED in version 2					
initializeATM	ATM	ATM	ATM	Synchronous	Asynchronous	5	4
setInitialCash	ATM	ATM	ATM	Synchronous	Synchronous	6	6
getATMAmount	OperationPanel	OperationPanel	OperationPanel	Synchronous	Synchronous	7	7

Table 27 ATMStartUp Sequence Diagrams Comparison Results

Message Name	Message Receiver	Changes	Changes/NMP
teurnON	Bank	0	0
notifyATM	Deposit	0	0
getATMStatus	Deposit	1	0.333
requestDollarAccount	Account	FULL	1
initializeATM	Account	1	0.333
setInitialCash	EnvelopeAcceptor	0	0
getATMAmount	OperationPanel	0	0
Sum			1.666
Instability			0.238
Stability			0.762

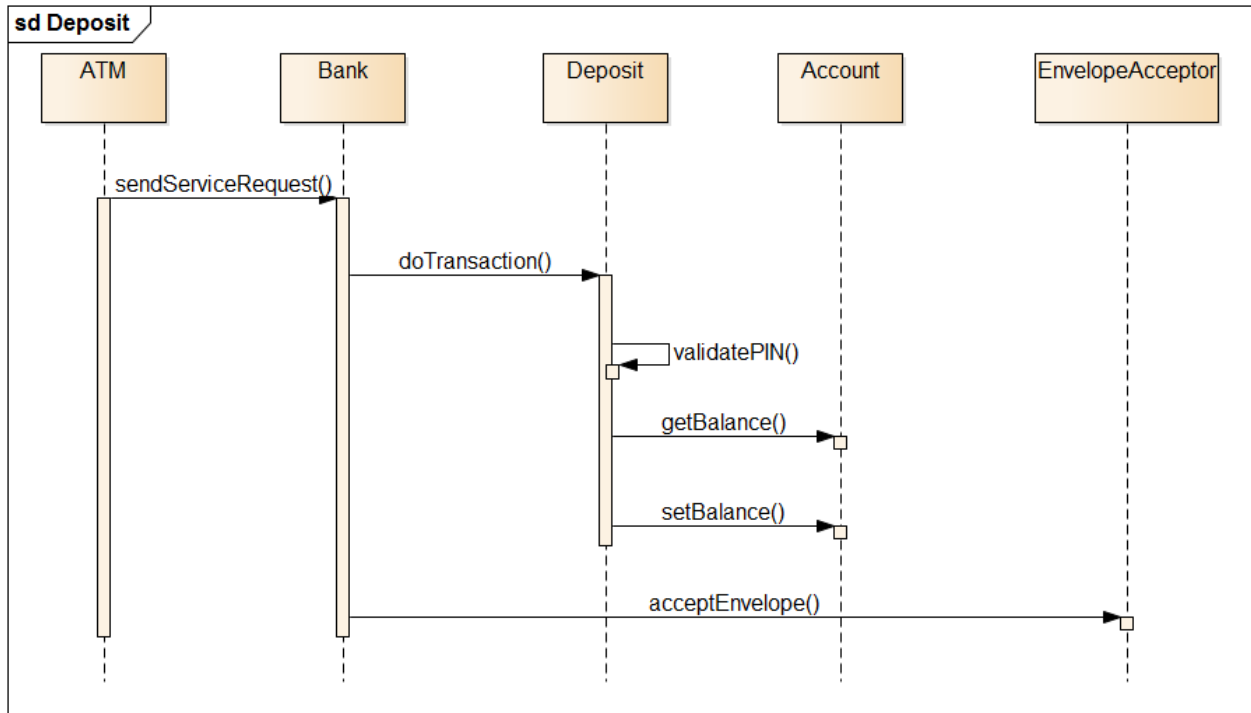


Figure 63 Deposit Sequence Diagram version 1

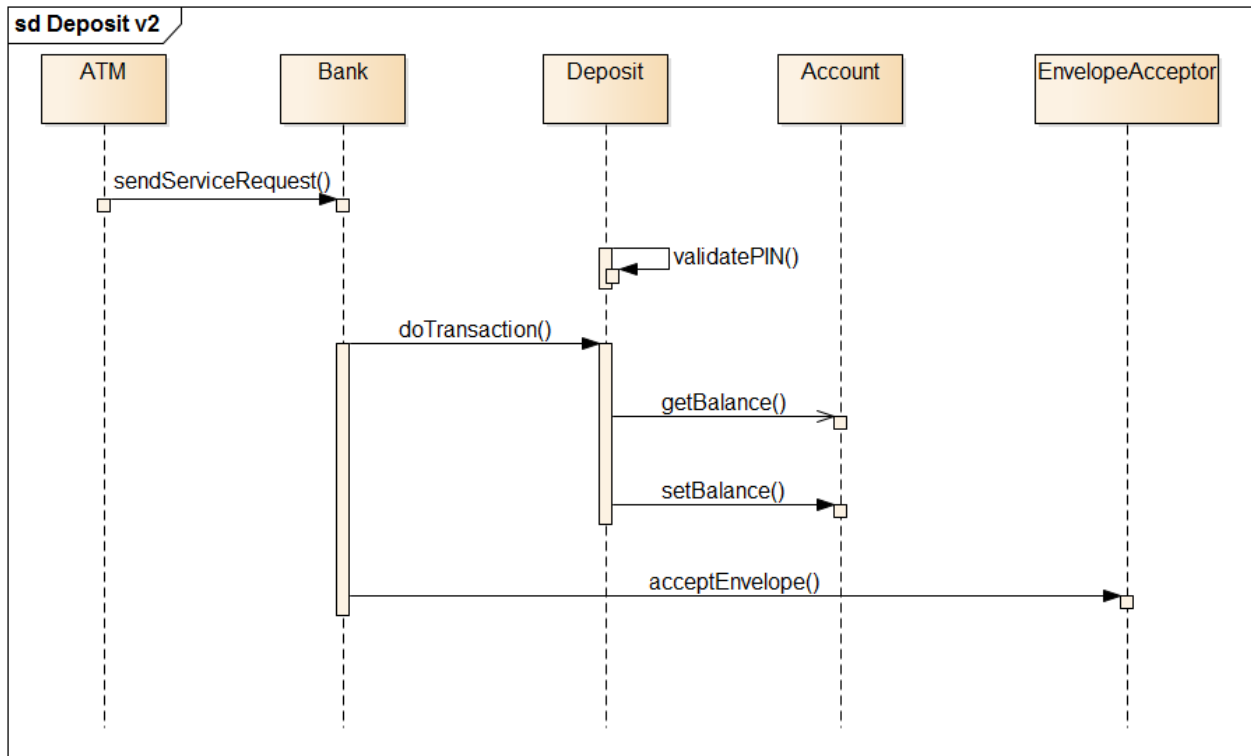


Figure 64 Deposit Sequence Diagram version 2

Table 28 Deposit Sequence Diagrams Comparison

The Identifier		Properties					
Message Name	Message Receiver	Message Caller		Message Type		Message Order	
		version 1	version 2	version 1	version 2	version 1	version 2
sendServiceRequest	Bank	ATM	ATM	Synchronous	Synchronous	1	1
doTransaction	Deposit	Bank	Bank	Synchronous	Synchronous	2	3
validatePIN	Deposit	Deposit	Deposit	Synchronous	Synchronous	3	2
getBalance	Account	Deposit	Deposit	Synchronous	Asynchronous	4	4
setBalance	Account	Deposit	Deposit	Synchronous	Synchronous	5	5
acceptEnvelope	EnvelopeAcceptor	Bank	Bank	Synchronous	Synchronous	6	6

Table 29 Deposit Sequence Diagrams Comparison Results

Message Name	Message Receiver	Changes	Changes / NMP
sendServiceRequest	Bank	0	0
doTransaction	Deposit	1	0.333
validatePIN	Deposit	1	0.333
getBalance	Account	1	0.333
setBalance	Account	0	0
acceptEnvelope	EnvelopeAcceptor	0	0
Sum			1
Instability			0.167
Stability			0.833

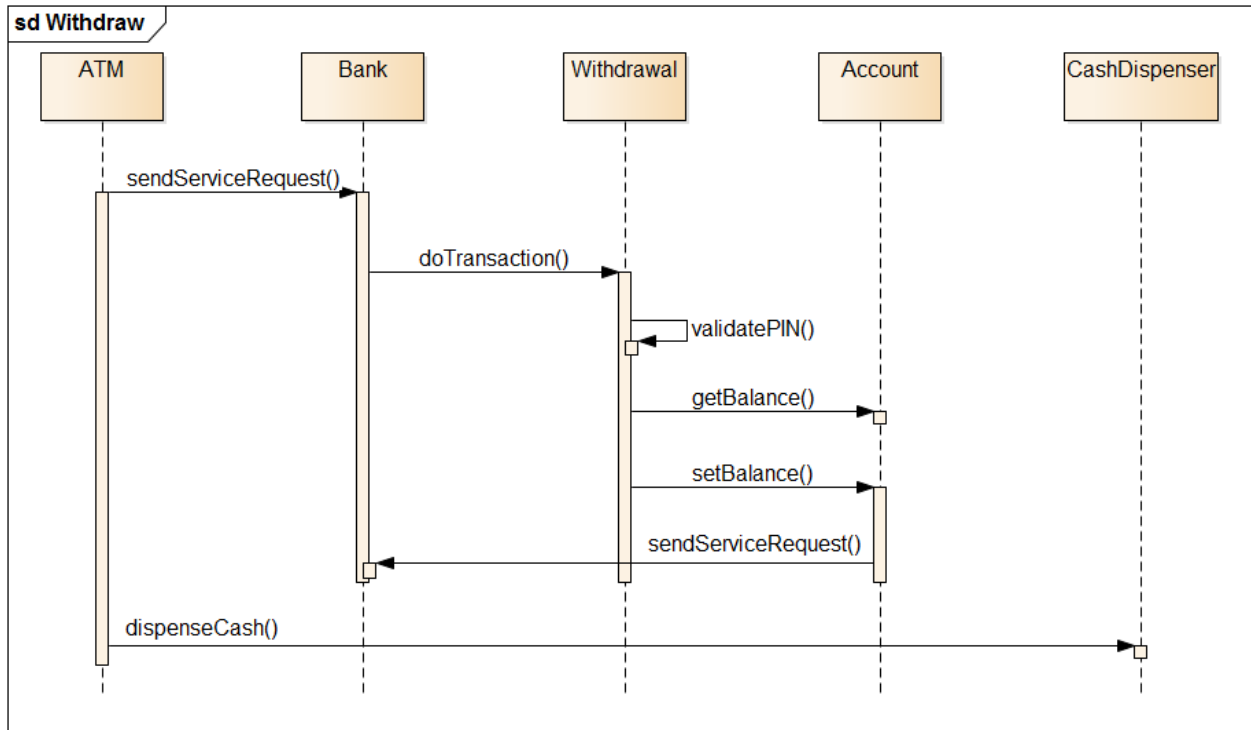


Figure 65 Withdrawal Sequence Diagram version 1

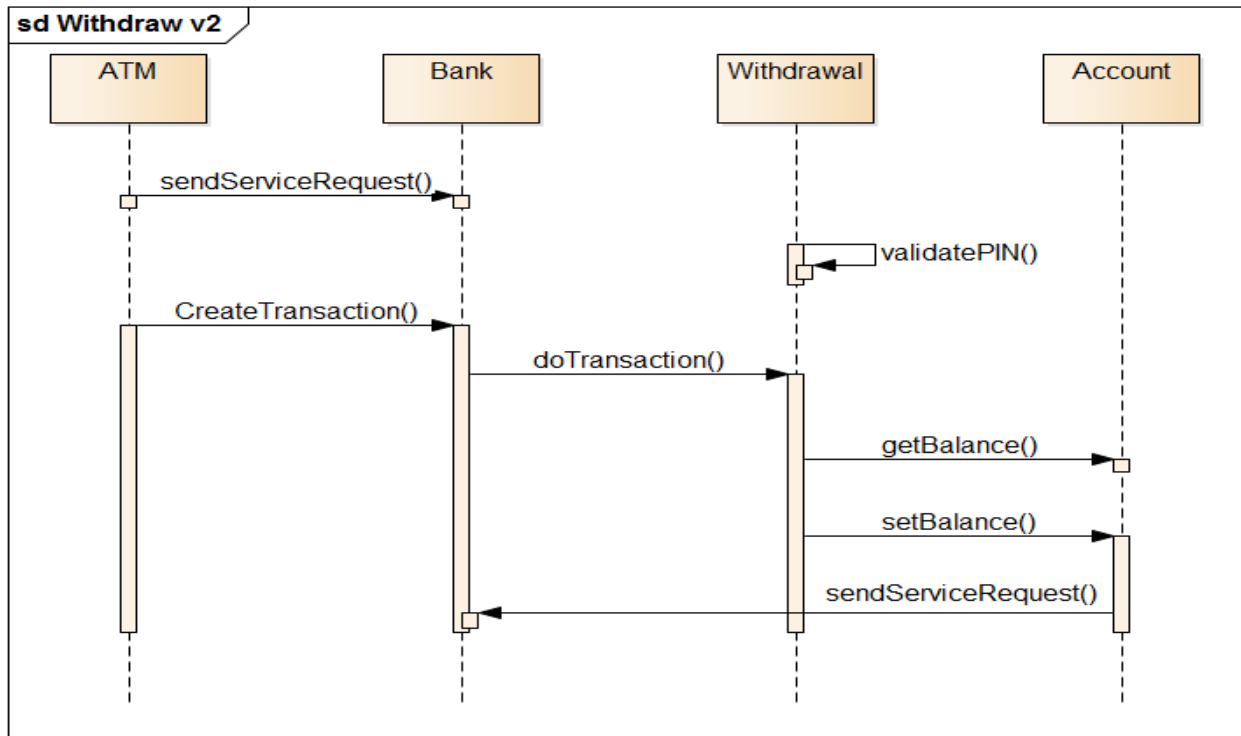


Figure 66 Withdrawal Sequence Diagram version 2

Table 30 Withdrawal Sequence Diagrams Comparison

The Identifier		Properties					
Message Name	Message Receiver	Message Caller		Message Type		Message Order	
		version 1	version 2	version 1	version 2	version 1	version 2
sendServiceRequest	Bank	ATM	ATM	Synchronous	Synchronous	1	1
doTransaction	Withdrawal	Bank	Bank	Synchronous	Synchronous	2	4
validatePIN	Withdrawal	Withdrawal	Withdrawal	Synchronous	Synchronous	3	2
getBalance	Account	Withdrawal	Withdrawal	Synchronous	Synchronous	4	5
setBalance	Account	Withdrawal	Withdrawal	Synchronous	Synchronous	5	6
sendServiceRequest	Bank	Account	Account	Synchronous	Synchronous	6	7
dispenseCash	CashDispenser	DELETED in version 2					

Table 31 Withdrawal Sequence Diagrams Results

Message Name	Message Receiver	Changes	Changes / NMP
sendServiceRequest	Bank	0	0
doTransaction	Withdrawal	1	0.333
validatePIN	Withdrawal	1	0.333
getBalance	Account	1	0.333
setBalance	Account	1	0.333
sendServiceRequest	Bank	1	0.333
dispenseCash	CashDispenser	FULL	1
Sum			2.666
Instability			0.38
Stability			0.62

9.2 Case Study 2: SCM

Supply chain management (SCM) application [55], we selected the Retailer subsystem. The Retailer's purpose is to present a Web service for a third party system. We used the class diagram and the existing sequence diagrams. The class diagram consists of nine classifiers. For the sequence diagram we selected three diagrams, and the average number of messages is three. Table 32 shows the experimental summary. Figure 67 and Figure 68 show the Retailer class diagram version 1 and version 2, respectively. Table 33 displays the comparison, and Table 34 shows the computation results. Sixty-six percent of version 1 of the class diagram remains in version 2.

Figure 69 and Figure 70 shows the first version and the second version of the Purchase sequence diagram respectively. The comparison is described in Table 35. The second version of the sequence diagram kept 76% of the first one, as shown in the Table 36 computations. Figure 71 and Figure 72 show the Replenish sequence diagram version 1 and version 2 respectively. The comparison is shown in Table 37. 44% of the first version is remaining at the second one as shown Table 38 computations. Figure 73 and Figure 74 shows the first version and the second version of Source sequence diagram, respectively. The comparison is described at Table 39. The second version of the sequence diagram kept 66% of the first one as shown in the Table 40 computations.

Table 32 SCM Case Study Summary

Diagram Type	System Name	Stability
Class Diagram	Retailer	0.462
Sequence Diagram	Purchase	0.666
Sequence Diagram	Replenish	0.444
Sequence Diagram	Source	0.666

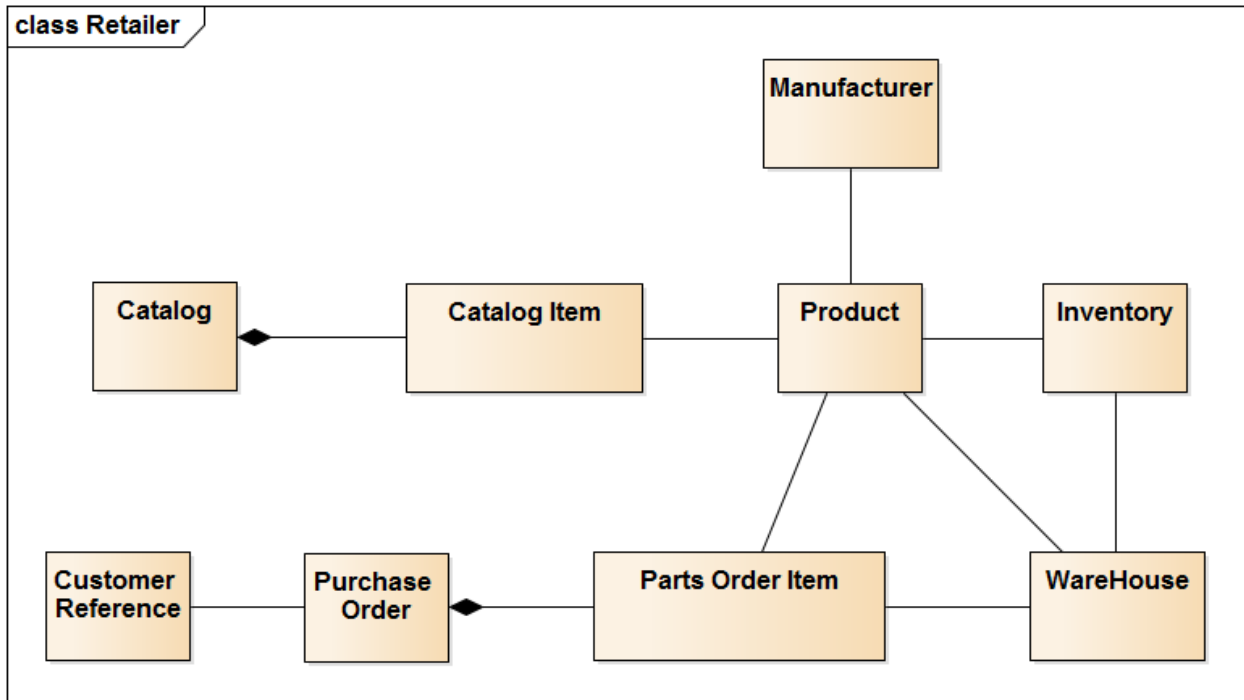


Figure 67 SCM Class Diagram version 1

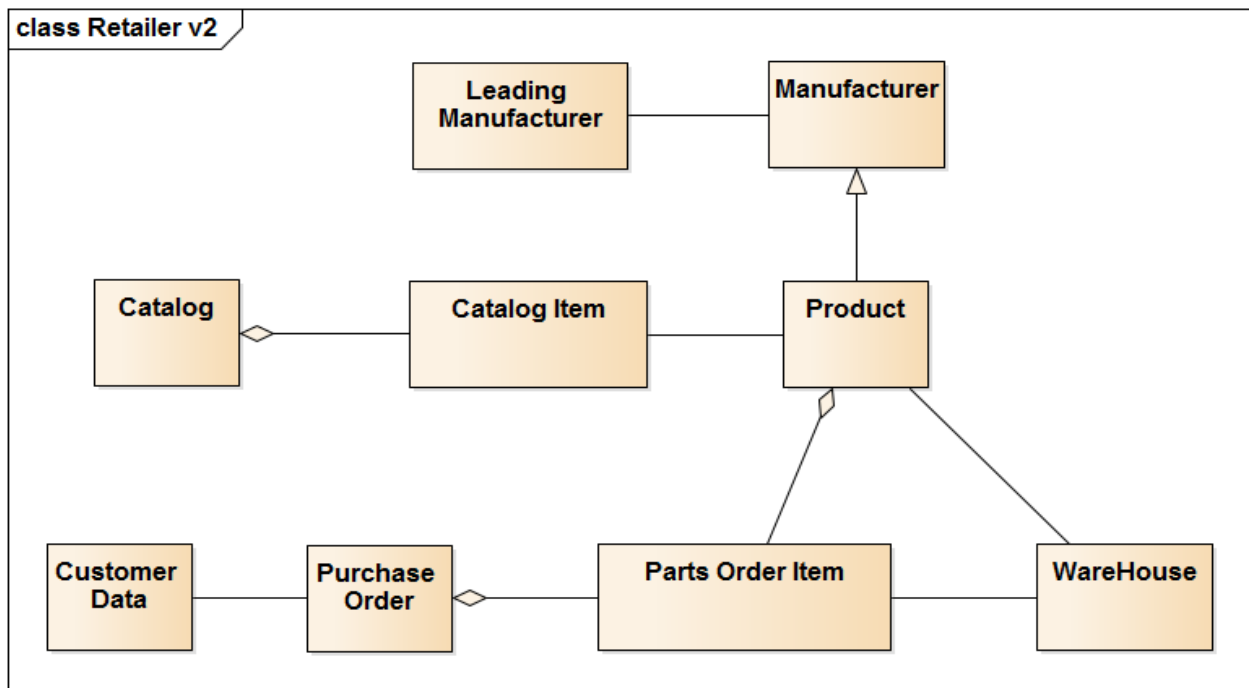


Figure 68 SCM Class Diagram version 2

Table 33 SCM Class Diagrams Comparison

Classifier Name	Version 1		Version 2	
	Classifier Type	Classifier Relationships	Classifier Type	Classifier Relationships
Manufacturer	Class	Product-ASO	Class	-
		-		LeadingManufacturer-ASO
Catalog	Class	CatalogItem-COM	Class	CatalogItem-AGG
Catalog Item	Class	Product-ASO	Class	Product-ASO
Product	Class	CatalogItem-ASO	Class	CatalogItem-ASO
		Inventory-ASO		Inventory-ASO
		Manufacturer-ASO		Manufacturer-INH
		PartsOrderItem-ASO		PartsOrderItem-AGG
		WareHouse-ASO		WareHouse-ASO
Inventory	Class	Product-ASO	DELETED	
		WareHouse-ASO		
Customer Reference	Class	PurchaseOrder-ASO	DELETED	
Purchase Order	Class	CustomerReference-ASO	Class	-
		PartsOrderItem-COM		PartsOrderItem-AGG
		-		CustomerData-ASO
Parts Order Item	Class	Product-ASO	Class	-
		WareHouse-ASO		WareHouse-ASO
WareHouse	Class	Inventory-ASO	Class	-
		Product-ASO		Product-ASO
		PartsOrderItem-ASO		PartsOrderItem-ASO

Table 34 SCM Class Diagrams Comparison Results

Classifier Name	Changes From version 1 to version 2	Number Of Changes	Number Of Unique Pairs	Changes / Unique
Manufacturer	Product-ASO => DELETED	2	3	0.666
	LeadingManufacturer-ASO => NEW			
Catalog	CatalogItem-COM => CatalogItem-AGG	1	2	0.5
Catalog Item	-	0	2	0
Product	Manufacturer-ASO => Manufacturer-INH	2	6	0.333
	PartsOrderItem-ASO => PartsOrderItem-AGG			
Inventory	DELETED	FULL	FULL	1
Customer Reference	DELETED	FULL	FULL	1
Purchase Order	CustomerReference-ASO => DELETED	3	4	0.75
	PartsOrderItem-COM => PartsOrderItem-AGG			
	CustomerData-ASO => NEW			
Parts Order Item	Product-ASO => DELETED	1	3	0.333
WareHouse	Inventory-ASO => DELETED	1	4	0.25
SUM				4.833
Instability				0.537
Stability				0.463

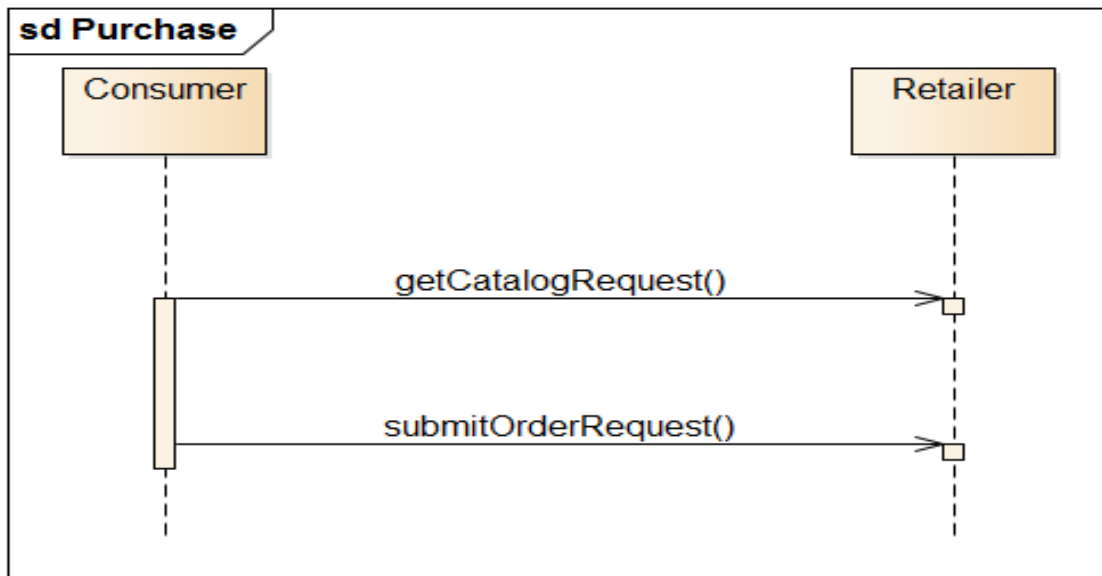


Figure 69 Purchase Sequence Diagram version 1

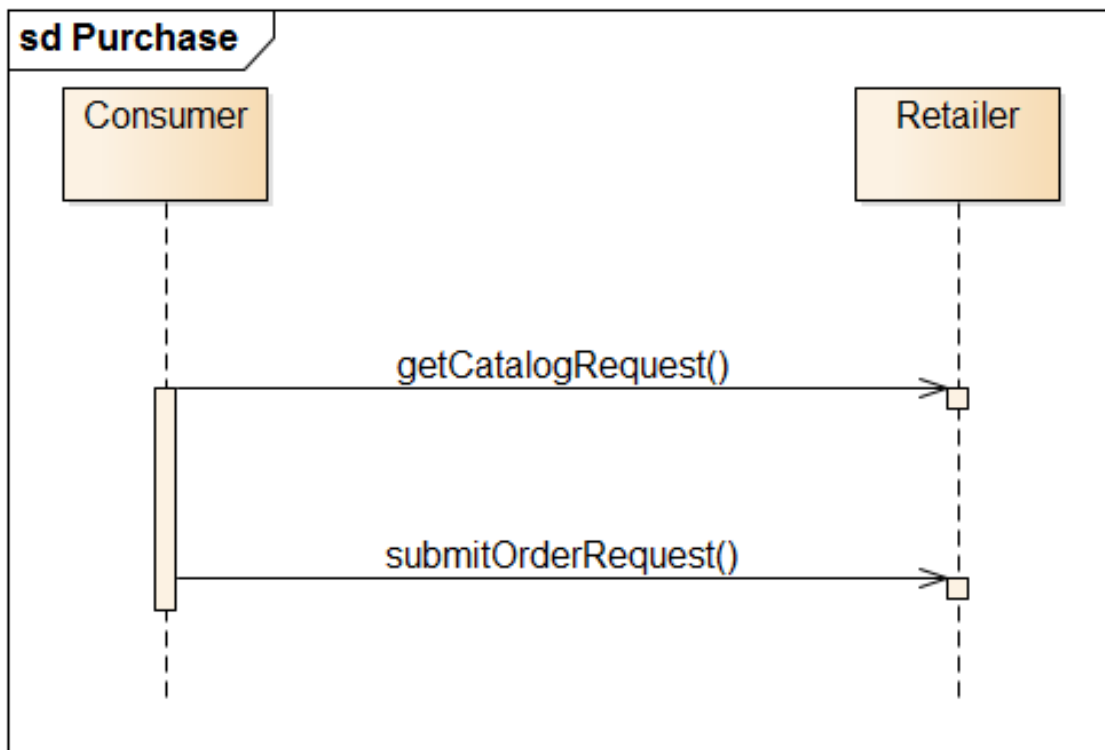


Figure 70 Purchase Sequence Diagram version 2

Table 35 Purchase Sequence Diagram Comparison

The Identifier		Properties					
Message Name	Message Receiver	Message Caller		Message Type		Message Order	
		version 1	version 2	version 1	version 2	version 1	version 2
getCatalogRequest	Retailer	Consumer	Consumer	Asynchronous	Synchronous	1	1
submitOrderRequest	Retailer	Consumer	Consumer	Asynchronous	Synchronous	2	2

Table 36 Purchase Sequence Diagram Comparison Results

Message Name	Message Receiver	Changes	Changes / NMP
getCatalogRequest	Retailer	1	0.333
submitOrderRequest	Retailer	1	0.333
Sum			0.666
Instability			0.333
Stability			0.667

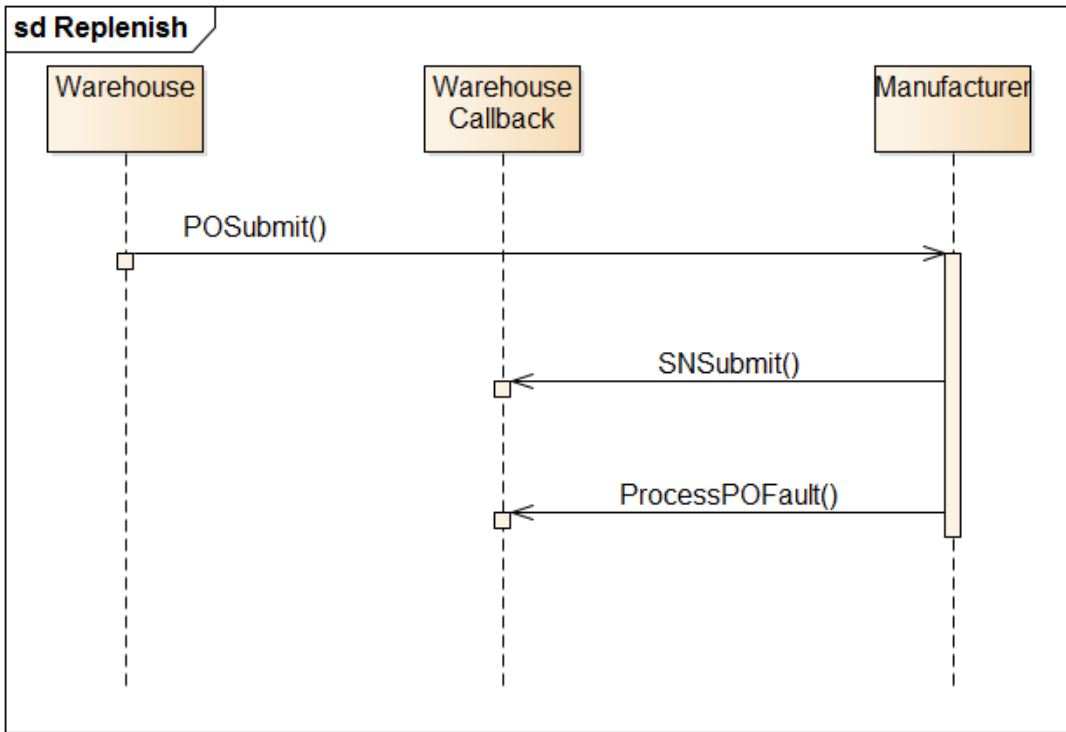


Figure 71 Replenish Sequence Diagram version 1

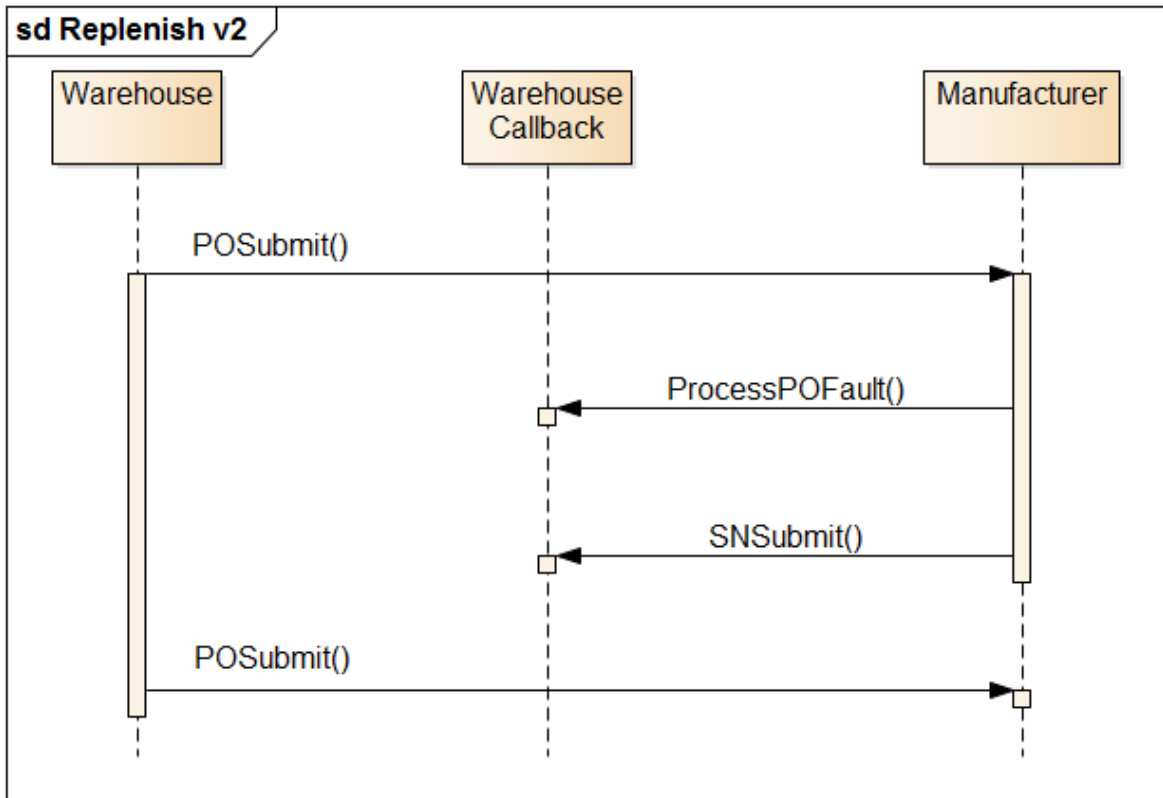


Figure 72 Replenish Sequence Diagram version 2

Table 37 Replenish Sequence Diagram Comparison

The Identifier		Properties					
Message Name	Message Receiver	Message Caller		Message Type		Message Order	
		version 1	version 2	version 1	version 2	version 1	version 2
POSubmit	Manufacturer	Warehouse	Warehouse	Asynchronous	Synchronous	1	1
SNSubmit	Warehouse Callback	Manufacturer	Manufacturer	Asynchronous	Synchronous	2	3
ProcessPOFault	Warehouse Callback	Manufacturer	Manufacturer	Asynchronous	Synchronous	3	2

Table 38 Replenish Sequence Diagram Comparison Results

Message Name	Message Receiver	Changes	Changes / NMP
POSubmit	Manufacturer	1	0.333
SNSubmit	Warehouse Callback	2	0.666
ProcessPOFault	Warehouse Callback	2	0.666
Sum			1.666
Instability			0.555
Stability			0.445

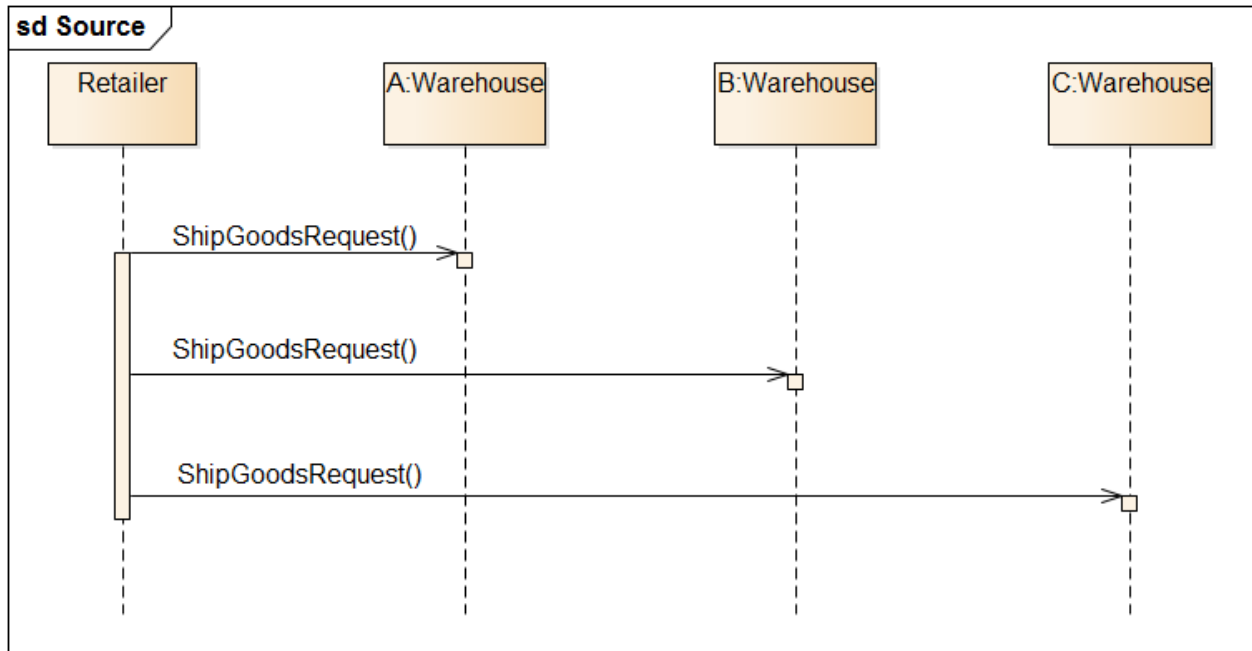


Figure 73 Source Sequence Diagram version 1

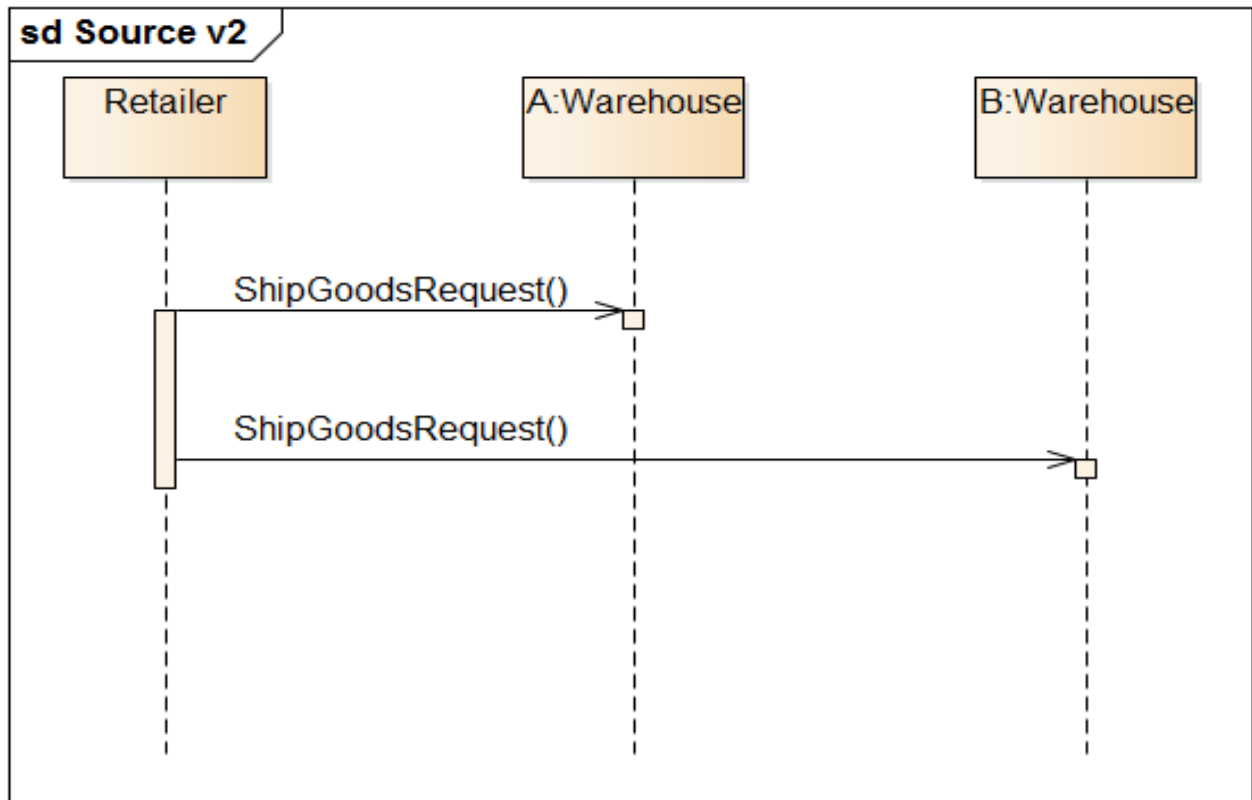


Figure 74 Source Sequence Diagram version 2

Table 39 Source Sequence Diagram Comparison

The Identifier		Properties					
Message Name	Message Receiver	Message Caller		Message Type		Message Order	
		version 1	version 2	version 1	version 2	version 1	version 2
ShipGoodsRequest	Warehouse	Retailer	Retailer	Asynchronous	Asynchronous	1	1
ShipGoodsRequest	Warehouse	Retailer	Retailer	Asynchronous	Asynchronous	2	2
ShipGoodsRequest	Warehouse	DELETED in version 2					

Table 40 Source Sequence Diagram Comparison Results

Message Name	Message Receiver	Changes	Changes / NMP
ShipGoodsRequest	Warehouse	0	0
ShipGoodsRequest	Warehouse	0	0
ShipGoodsRequest	Warehouse	Full	1
Sum			1
Instability			0.333
Stability			0.667

9.3 Case Study 3: ORA

For the On Road Assistance (ORA) [56], we used the use case diagram. The diagram contains 13 use cases and five different actors.

Figure 75 and Figure 76 show Retailer use case diagram version 1 and version 2, respectively. Table 41 displays the comparison, and Table 42 shows the computation results. Seventy-eight percent of use case version 1 remains in version 2.

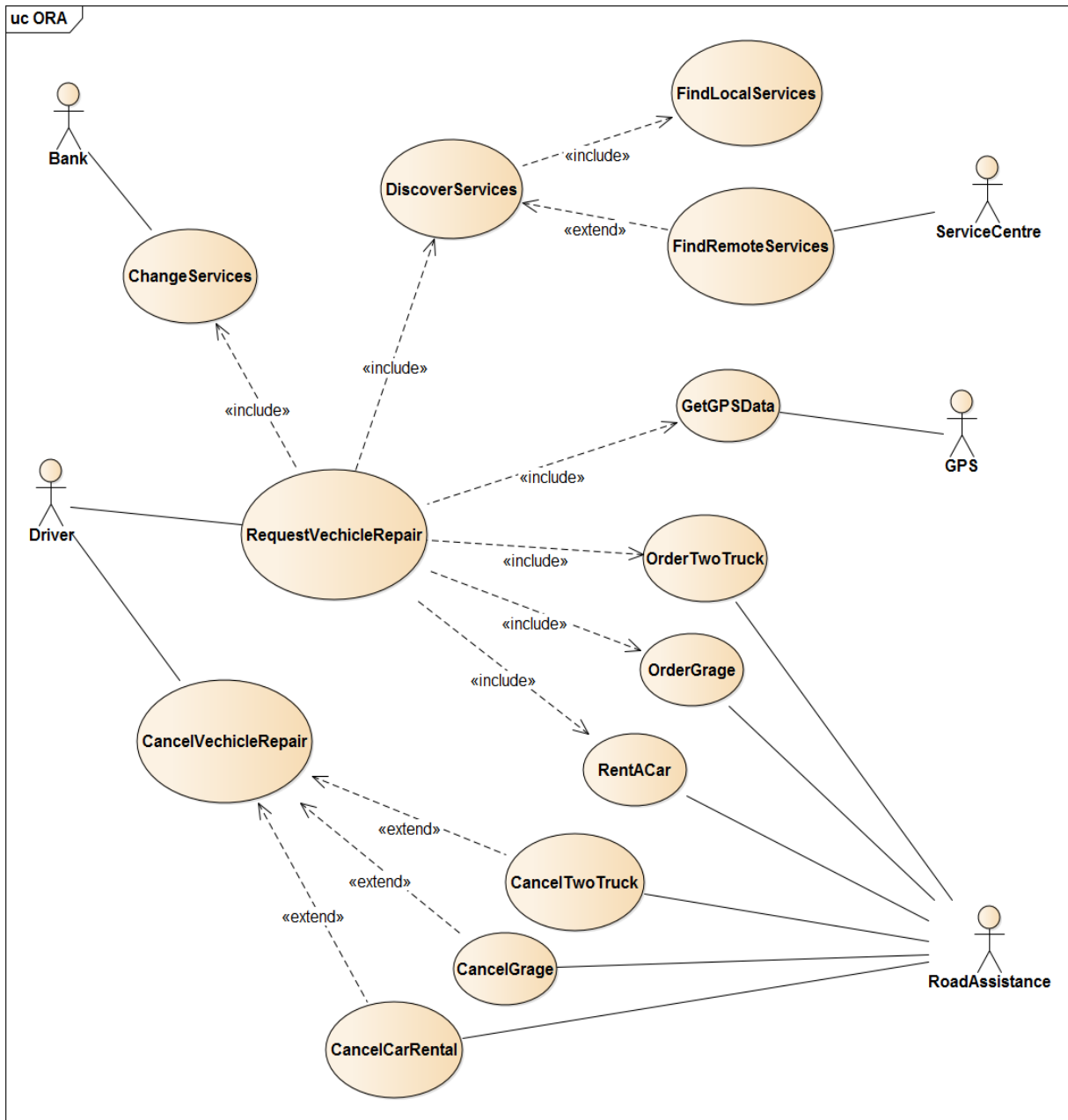


Figure 75 ORA Use Case Diagram version 1

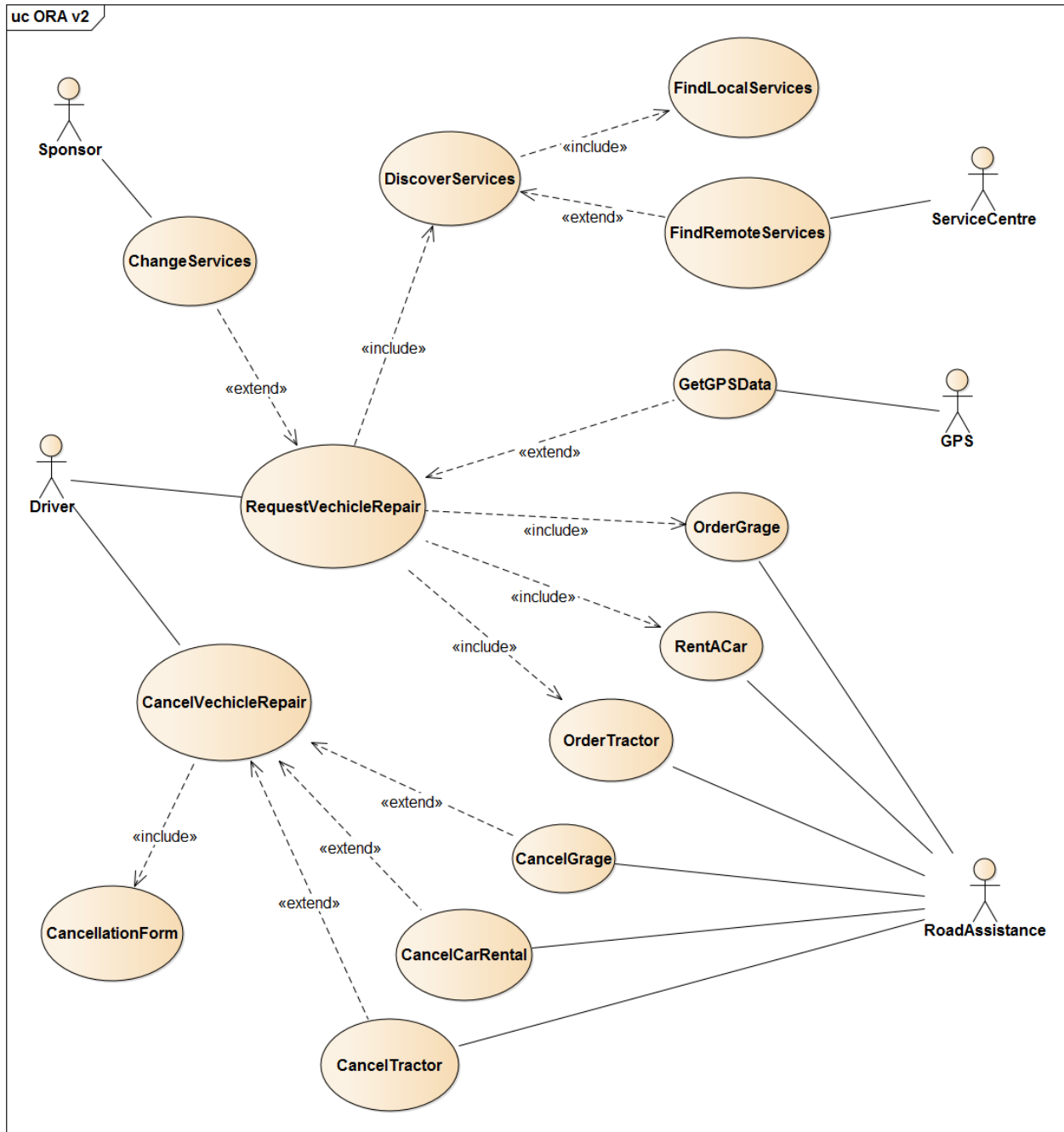


Figure 76 ORA Use Case Diagram version 2

Table 41 ORA Use Case Diagram Comparison

Identifier Name	Version 1		Version 2	
	Identifier Type	Identifier Relationships	Identifier Type	Identifier Relationships
Bank	DELETED - Change the name to Sponsor			
ChangeServices	Use Case	Bank-ASO	Use Case	-
		-		RequestVechicleRepair-EX
		-		Sponsor-ASO
DiscoverServices	Use Case with Extension Point	FindLocalServies-INC	Use Case with Extension Point	FindLocalServies-INC
FindLocalServies	Use Case	-	Use Case	-
FindRemoteServices	Use Case	ServiceCentre-ASO	Use Case	ServiceCentre-ASO
		DiscoverServices-EX		DiscoverServices-EX
ServiceCentre	Actor	FindRemoteServices-ASO	Actor	FindRemoteServices-ASO
Driver	Actor	RequestVechicleRepair-ASO	Actor	RequestVechicleRepair-ASO
		CancelVechicleRepair-ASO		CancelVechicleRepair-ASO
RequestVechicleRepair	Use Case	ChangeServices-INC	Use Case with Extension Point	-
		DiscoverServices-INC		DiscoverServices-INC
		GetGPSData-INC		-
		OrderTwoTruck-INC		OrderTwoTruck-INC
		OrderGrage-INC		OrderGrage-INC

		RentACar-INC		RentACar-INC
CancelVechicleRepair	Use Case with Extension Point	Driver-ASO	Use Case with Extension Point	Driver-ASO
		-		CancellationForm-INC
GetGPSData	Use Case	GPS-ASO	Use Case	GPS-ASO
		-		RequestVechicleRepair-EX
GPS	Actor	GetGPSData-ASO	Actor	GetGPSData-ASO
OrderTwoTruck	DELETED - Change the name to OderTractor			
OrderGrage	Use Case	RoadAssistance-ASO	Use Case	RoadAssistance-ASO
RentACar	Use Case	RoadAssistance-ASO	Use Case	RoadAssistance-ASO
CancelTwoTruck	Use Case	RoadAssistance-ASO	Use Case	RoadAssistance-ASO
		CancelVechicleRepair-EX		CancelVechicleRepair-EX
CancelGrage	Use Case	RoadAssistance-ASO	Use Case	RoadAssistance-ASO
		CancelVechicleRepair-EX		CancelVechicleRepair-EX
CancelCarRental	Use Case	RoadAssistance-ASO	Use Case	RoadAssistance-ASO
		CancelVechicleRepair-EX		CancelVechicleRepair-EX
RoadAssistance	Actor	OrderTwoTruck-ASO	Actor	OrderTwoTruck-ASO
		OrderGrage-ASO		OrderGrage-ASO
		RentACar-ASO		RentACar-ASO
		CancelTwoTruck-ASO		CancelTwoTruck-ASO
		CancelGrage-ASO		CancelGrage-ASO
		CancelCarRental-ASO		CancelCarRental-ASO

Table 42 ORA Class Diagram Comparison Results

Identifier Name	Changes From version 1 to version 2	Number Of Changes	Number Of Unique Pairs	Changes/Unique
Bank	DELETED	FULL	-	1
ChangeServices	Bank-ASO => DELETED	3	4	0.75
	RequestVechicleRepair-EX => NEW			
	Sponsor-ASO => NEW			
DiscoverServices	-	0	-	0
FindLocalServies	-	0	-	0
FindRemoteServices	-	0	-	0
ServiceCentre	-	0	-	0
Driver	-	0	-	0
RequestVechicleRepair	Use Case => Use Case with Extension Point	3	7	0.428
	ChangeServices-INC => DELETED			
	GetGPSData-INC => DELETED			
CancelVechicleRepair	CancellationForm-INC => NEW	1	3	0.333
GetGPSData	RequestVechicleRepair-EX => NEW	1	3	0.333

GPS	-	0	-	0
OrderTwoTruck	DELETED	FULL	-	1
OrderGrage	-	0	-	0
RentACar	-	0	-	0
CancelTwoTruck	-	0	-	0
CancelGrage	-	0	-	0
CancelCarRental	-	0	-	0
RoadAssistance	-	0	-	0
SUM				3.845
Instability				0.214
Stability				0.786

9.4 Case Study 4: O-RED System

The Online Real Estate Directory (O-RED) provides an online directory of the Real Estate offers to serve the end user. We used the user management class diagram, which contains 11 classifiers.

Figure 77 and Figure 78 show Retailer class diagram version 1 and version 2, respectively. Table 43 displays the comparison, and Table 44 shows the computation results. Seventy-one percent of version 1 of the class diagram remains in version 2.

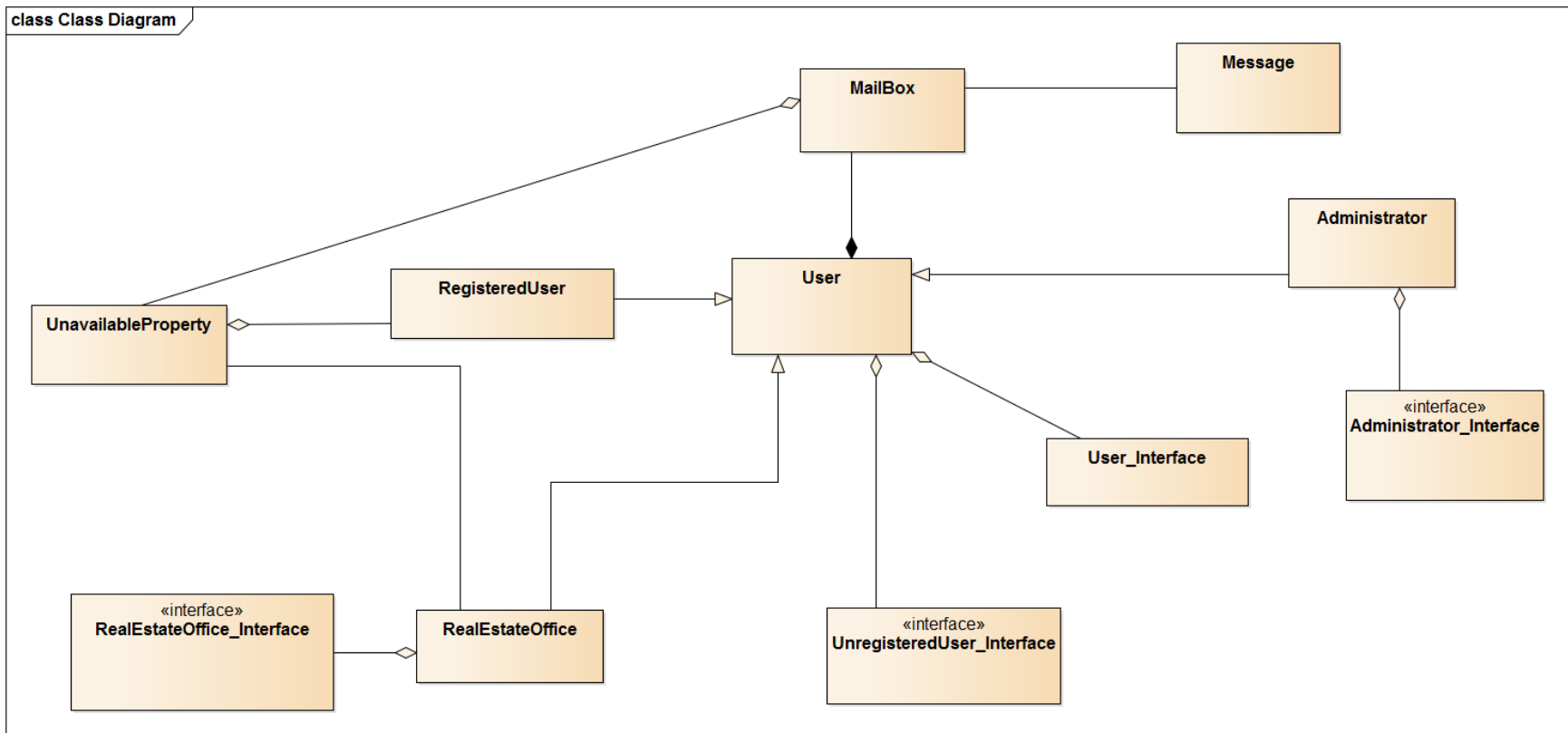


Figure 77 O-RED Class Diagram version 1

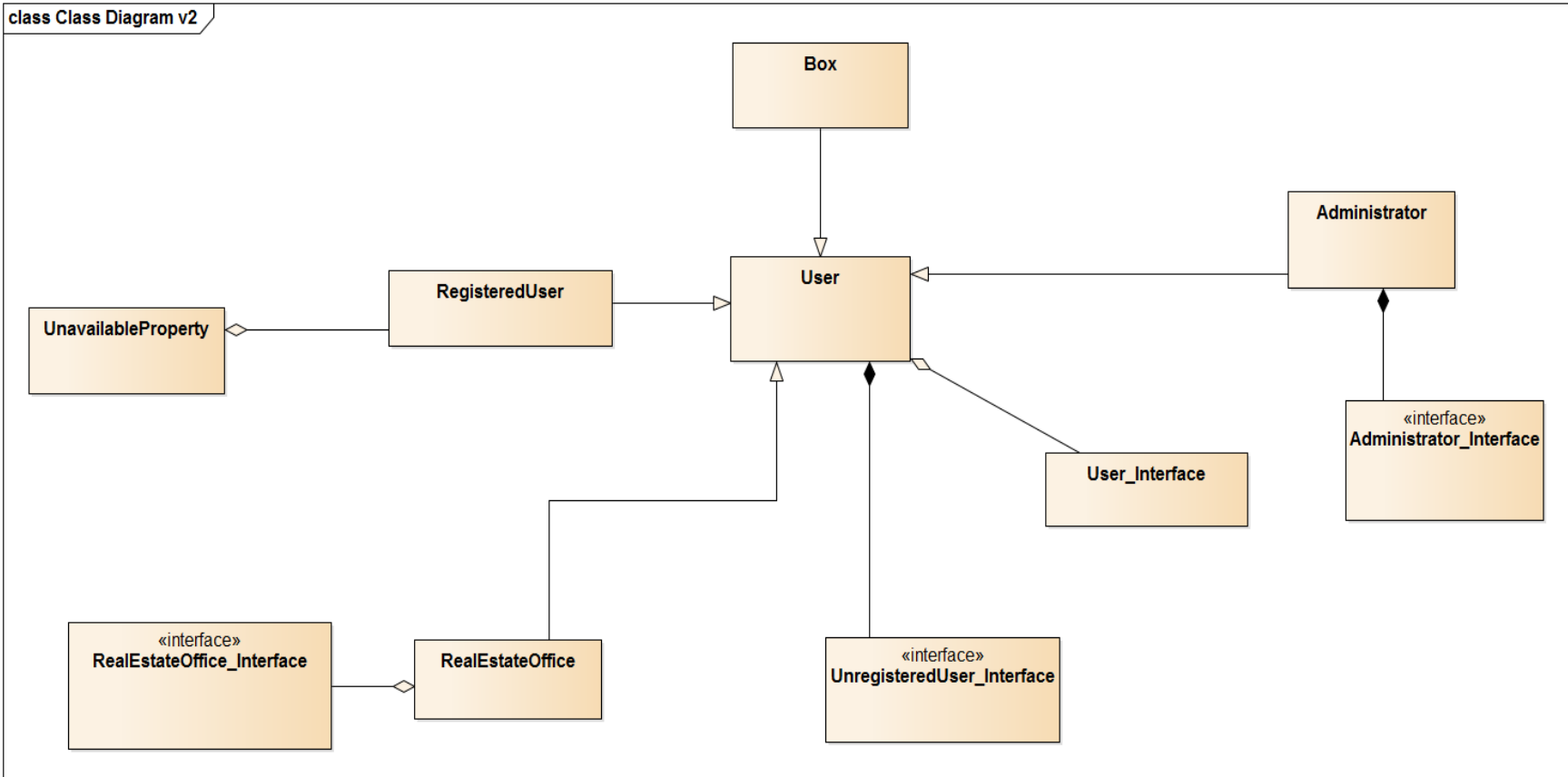


Figure 78 O-RED Class Diagram version 2

Table 43 O-RED Class Diagram Comparison

Classifier Name	Version 1		Version 2	
	Classifier Type	Classifier Relationships	Classifier Type	Classifier Relationships
MailBox	Class	Message-ASO	DELETED	
		UnavailableProperty-AGG		
Message	Class	MailBox-ASO	DELETED	
UnavailableProperty	Class	RegisteredUser-AGG	Class	RegisteredUser-AGG
		RealEstateOffice-ASO		-
RegisteredUser	Class	User-INH	Class	User-INH
User	Class	MailBox-COM	Class	-
		User_Interface-AGG		User_Interface-AGG
		UnregisteredUser_Interface-AGG		UnregisteredUser_Interface-AGG
Administrator	Class	User-INH	Class	User-INH
		Administrator_Interface-AGG		Administrator_Interface-COM
Administrator_Interface	Interface	-	Interface	-
User_Interface	Interface	-	Interface	-
UnregisteredUser_Interface	Interface	-	Interface	-
RealEstateOffice	Class	User-INH	Class	User-INH
		UnavailableProperty-ASO		-
		RealEstateOffice_Interface-		RealEstateOffice_Interface-AGG
RealEstateOffice_Interface	Interface	-	Interface	-

Table 44 O-RED Class Diagram Comparison Results

Classifier Name	Changes From version 1 to version 2	Number Of Changes	Number Of Unique Pairs	Changes / Unique
MailBox	DELETED	FULL	FULL	1
Message	DELETED	FULL	FULL	1
UnavailableProperty	RealEstateOffice-ASO => DELETED	1	3	0.333
RegisteredUser	-	0	-	0
User	MailBox-COM => DELETED	1	4	0.25
Administrator	Administrator_Interface-AGG => Administrator_Interface-COM	1	3	0.333
Administrator_Interface	-	0	-	0
User_Interface	-	0	-	0
UnregisteredUser_Interface	-	0	-	0
RealEstateOffice	UnavailableProperty-ASO => UnavailableProperty-AGG	1	4	0.25
RealEstateOffice_Interface	-	0	-	0
			SUM	3.16
			Instability	0.287
			Stability	0.713

9.5 Case Study 5: HOSS System

The Hajj Online Services System (HOSS) is an online service of Hajj management. We used the use case diagram of the Communication Management Subsystem. It contains nine use cases and three actors.

Figure 79 and Figure 80 show Retailer use case diagram version 1 and version 2, respectively. Table 45 displays the comparison, and Table 46 shows the computation results. Fifty-three percent of version 1 of the use case diagram remains in version 2.

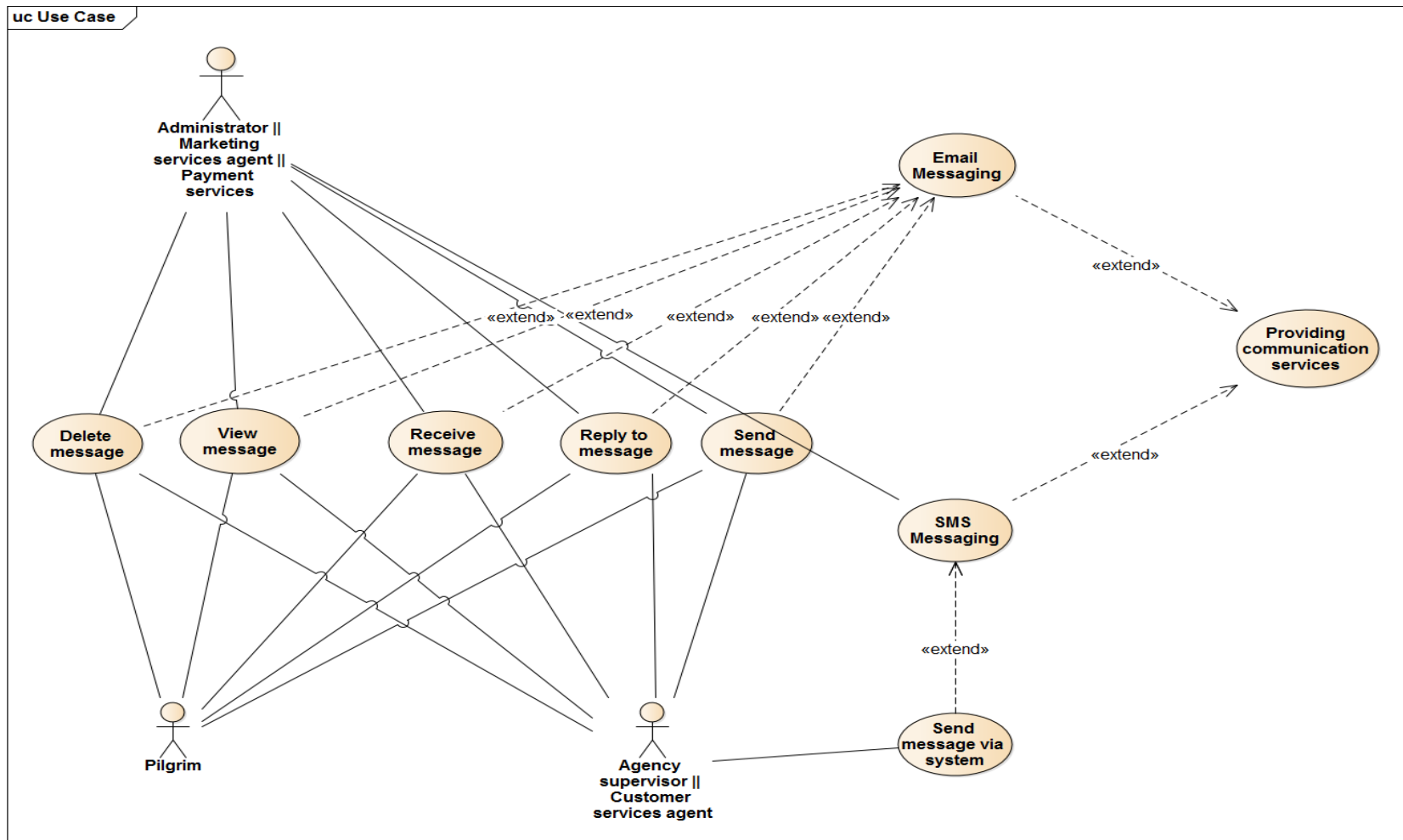


Figure 79 HOSS Use Case Diagram version 1

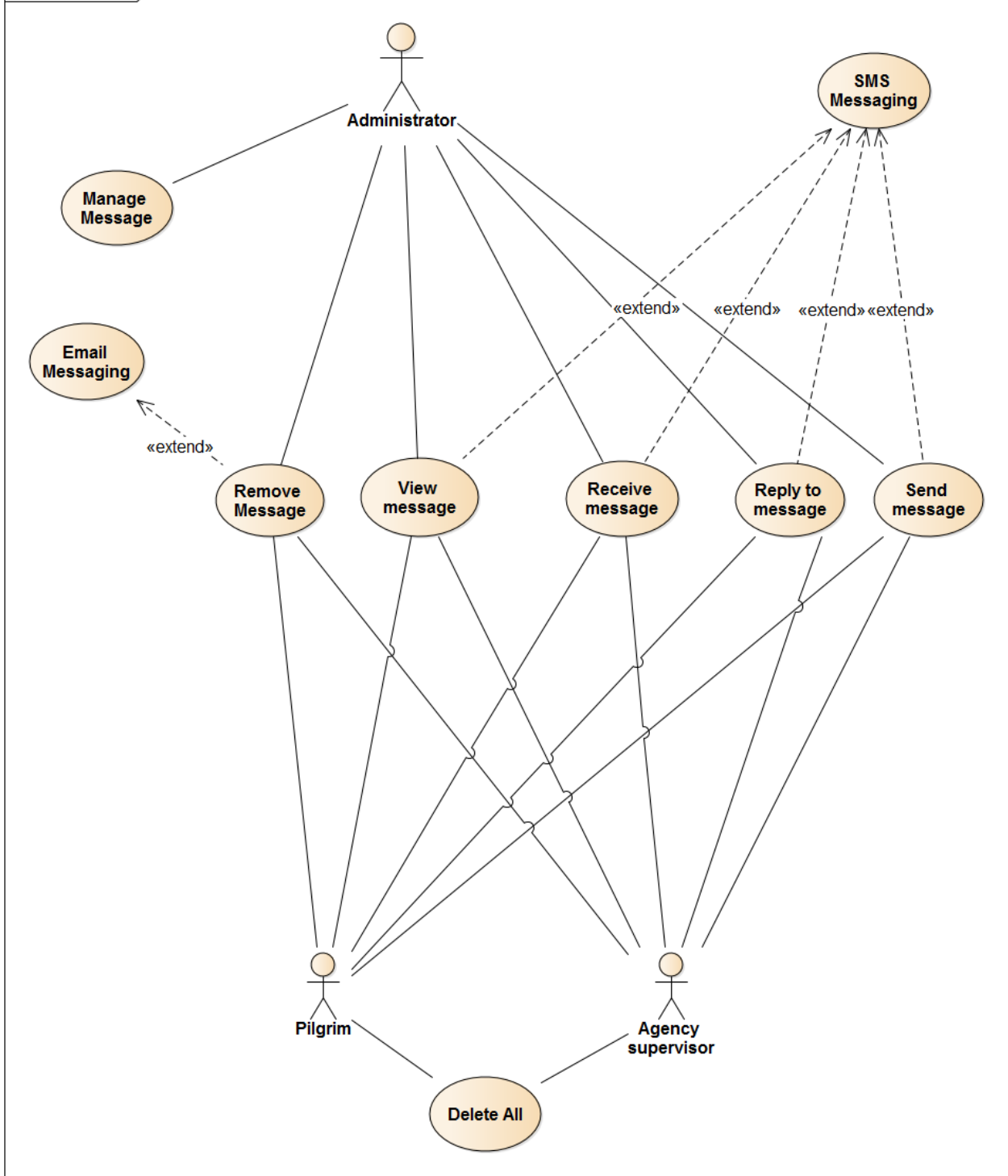


Figure 80 HOSS Use Case Diagram version 2

Table 45 HOSS Use Case Diagram Comparison

Identifier Name	Version 1		Version 2	
	Identifier Type	Identifier Relationships	Identifier Type	Identifier Relationships
Administrator	Actor	Delete message-ASO	Actor	-
		View message-ASO		View message-ASO
		Receive message-ASO		Receive message-ASO
		Reply to message-ASO		Reply to message-ASO
		Send message-ASO		Send message-ASO
		SMS Messaging-ASO		-
		-		Manage messages -ASO
		-		Remove Message - ASO
Delete message	DELETED - Change the name to Remove Message			
View message	Use Case	Administrator-ASO	Use Case	Administrator-ASO
		Pilgrim-ASO		Pilgrim-ASO
		Agency supervisor-ASO		Agency supervisor-ASO
		Email Messaging-EX		-
		-		SMS Messaging-EX
Receive message	Use Case	Administrator-ASO	Use Case	Administrator-ASO
		Pilgrim-ASO		Pilgrim-ASO
		Agency supervisor-ASO		Agency supervisor-ASO
		Email Messaging-EX		-
		-		SMS Messaging-EX
Reply to message	Use Case	Administrator-ASO	Use Case	Administrator-ASO
		Pilgrim-ASO		Pilgrim-ASO
		Agency supervisor-ASO		Agency supervisor-ASO
		Email Messaging-EX		-
		-		SMS Messaging-EX
Send message	Use Case	Administrator-ASO	Use Case	Administrator-ASO
		Pilgrim-ASO		Pilgrim-ASO

		Agency supervisor-ASO		Agency supervisor-ASO
		Email Messaging-EX		-
		-		SMS Messaging-EX
Pilgrim	Actor	Delete message-ASO	Actor	-
		View message-ASO		View message-ASO
		Receive message-ASO		Receive message-ASO
		Reply to message-ASO		Reply to message-ASO
		Send message-ASO		Send message-ASO
		-		Delete All - ASO
		-		Remove Message - ASO
Agency supervisor	Actor	Delete message-ASO	Actor	-
		View message-ASO		View message-ASO
		Receive message-ASO		Receive message-ASO
		Reply to message-ASO		Reply to message-ASO
		Send message-ASO		Send message-ASO
		Send message via system-ASO		Send message via system-ASO
		-		Delete All - ASO
-	Remove Message - ASO			
SMS Messaging	Use Case with Extension Point	Administrator-ASO	Use Case with Extension Point	Administrator-ASO
		Providing communication services-EX		-
Email Messaging	Use Case with Extension Point	Providing communication services-EX	Use Case with Extension Point	-
Send message via system	DELETED			
Providing communication services	DELETED			

Table 46 HOSS Use Case Diagram Comparison Results

Identifier Name	Changes From version 1 to version 2	Number Of Changes	Number Of Unique Pairs	Changes /Unique
Administrator	Delete message-ASO => DELETED	4	9	0.444
	SMS Messaging-ASO => DELETED			
	Manage messages -ASO => NEW			
	Remove Message - ASO => NEW			
Delete message	DELETED	FULL	-	1
View message	Email Messaging-EX => DELETED	2	6	0.333
	SMS Messaging-EX => NEW			
Receive message	Email Messaging-EX => DELETED	2	6	0.333
	SMS Messaging-EX => NEW			
Reply to message	Email Messaging-EX => DELETED	2	6	0.333
	SMS Messaging-EX => NEW			
Send message	Email Messaging-EX => DELETED	2	6	0.333
	SMS Messaging-EX => NEW			
Pilgrim	Delete message-ASO => DELETED	3	8	0.375
	Delete All - ASO => NEW			
	Remove Message - ASO => NEW			
Agency supervisor	Delete message-ASO => DELETED	3	9	0.333
	Delete All - ASO => NEW			

	Remove Message - ASO => NEW			
SMS Messaging	Providing communication services-EX => DELETED	1	3	0
Email Messaging	Providing communication services-EX => DELETED	1	2	0.5
Send message via system	DELETED	FULL	-	1
Providing communication services	DELETED	FULL	-	1
SUM				5.541
Instability				0.462
Stability				0.538

9.6 Case Study 6: ESAP System

The Electronic Students' Academic Portfolio, (ESAP), is an application used to help the DAD department to achieve their goals and do their work more efficiently with less paper work. We used the existing use case diagram that consists of 13 use cases and two actors.

Figure 81 and Figure 82 show Retailer use case diagram version 1 and version 2, respectively. Table 47 displays the comparison, and Table 48 shows the computation results. Sixty-one percent of version 1 of the use case diagram remains in version 2.

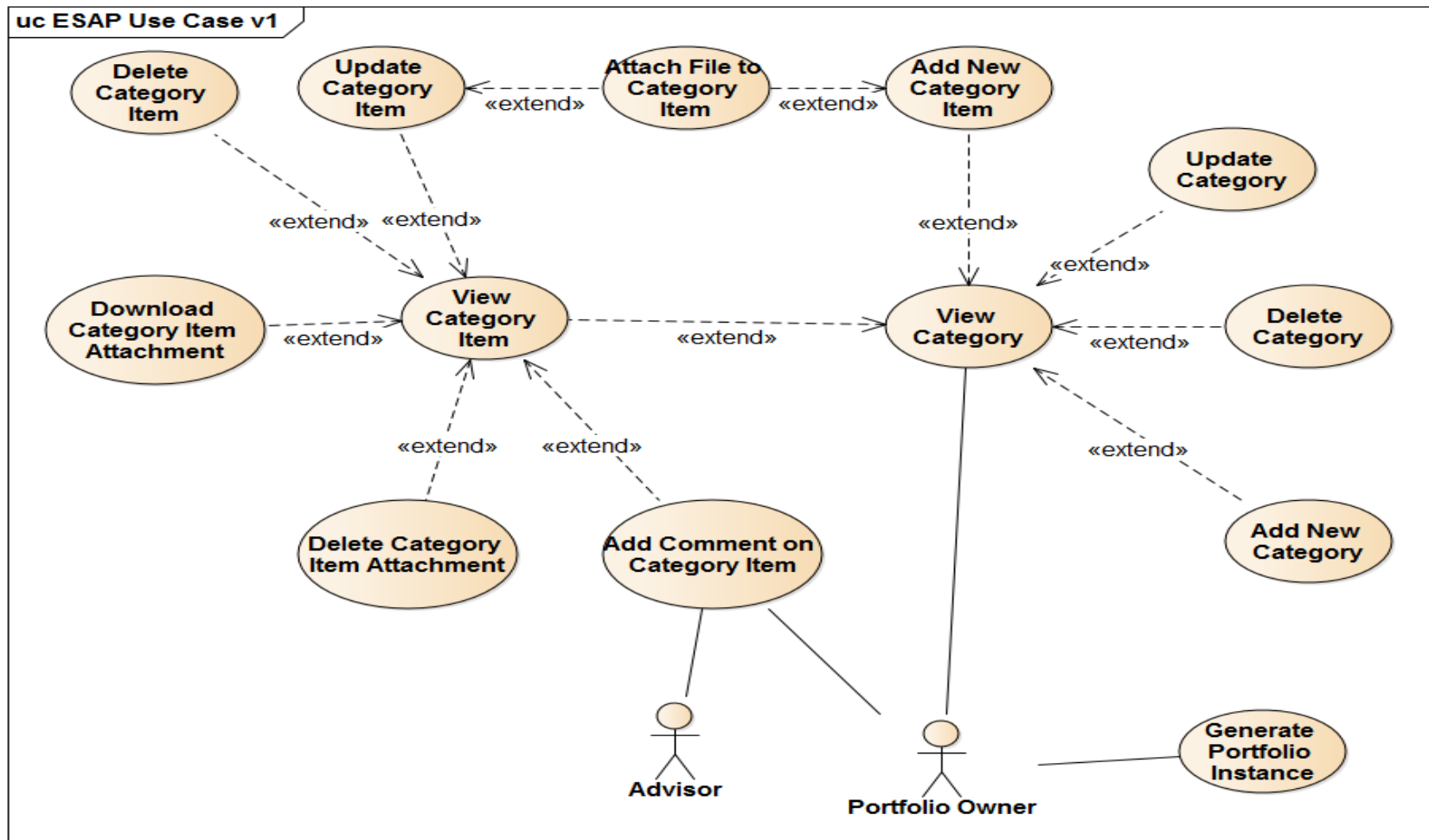


Figure 81 ESAP Use Case Diagram version 1

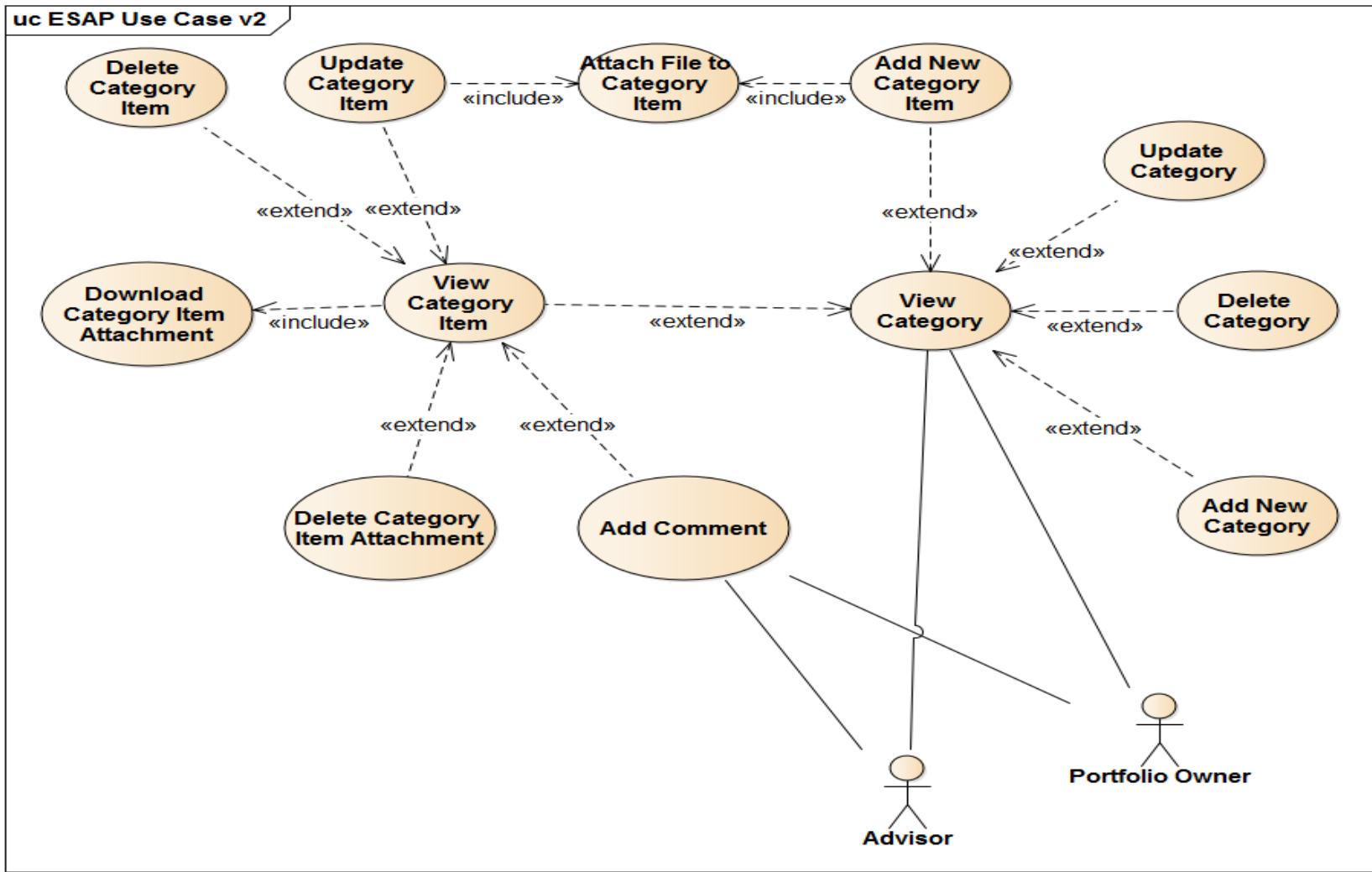


Figure 82 ESAP Use Case Diagram version 2

Table 47 ESAP Use Case Diagram Comparison

Identifier Name	Version 1		Version 2	
	Identifier Type	Identifier Relationships	Identifier Type	Identifier Relationships
Delete Category Item	Use Case	View Category Item - EX	Use Case	View Category Item - EX
Update Category Item	Use Case with Extension Point	View Category Item - EX	Use Case	View Category Item - EX
		-		Attach File to Category Item - INC
Attach File to Category Item	Use Case	Update Category Item - EX	Use Case	-
		Add New Category Item - EX		-
Add New Category Item	Use Case with Extension Point	View Category - EX	Use Case	View Category - EX
		-		Attach File to Category Item - INC
Download Category Item Attachment	Use Case	View Category Item - EX	Use Case	-
View Category Item	Use Case with Extension Point	View Category - EX	Use Case with Extension Point	View Category - EX
		-		Download Category Item Attachment - INC
View Category	Use Case with Extension Point	Portfolio Owner - ASO	Use Case with Extension Point	Portfolio Owner - ASO
		-		Advisor - ASO
Update Category	Use Case	View Category - EX	Use Case	View Category - EX
Delete Category	Use Case	View Category - EX	Use Case	View Category - EX
Delete Category Item Attachment	Use Case	View Category Item - EX	Use Case	View Category Item - EX
Add Comment on Category Item	DELETED - Change the name to Add Comment			

Add New Category	Use Case	View Category Item - EX	Use Case	View Category Item - EX
Generate Portfolio Instance	DELETED			
Advisor	Actor	Add Comment on Category Item - EX	Actor	-
		-		View Category - ASO
		-		Add Comment - ASO
Portfolio Owner	Actor	Add Comment on Category Item - EX	Actor	-
		View Category - EX		View Category - EX
		Generate Portfolio Instance - EX		-
		-		Add Comment - ASO

Table 48 ESAP Use Case Diagram Comparison Results

Identifier Name	Changes From version 1 to version 2	Number Of Changes	Number Of Unique Pairs	Changes / Unique
Delete Category Item	-	0	-	0
Update Category Item	Attach File to Category Item - INC => NEW	1	3	0.333
Attach File to Category Item	Update Category Item - EX => DELETED	2	3	0.666
	Add New Category Item - EX => DELETED			
Add New Category Item	Attach File to Category Item - INC => NEW	1	3	0.333
Download Category Item Attachment	View Category Item - EX => DELETED	1	2	0.5
View Category Item	Download Category Item Attachment - INC => NEW	1	3	0.333
View Category	Advisor - ASO => NEW	1	3	0.333
Update Category	-	0	-	0
Delete Category	-	0	-	0
Delete Category Item Attachment	-	0	-	0
Add Comment on Category Item	DELETED	FULL	-	1
Add New Category	-	0	-	0
Generate Portfolio Instance	DELETED	FULL	-	1
Advisor	Add Comment on Category Item - EX => DELETED	3	4	0.75
	View Category - ASO => NEW			
	Add Comment - EX => NEW			
Portfolio Owner	Add Comment on Category Item - EX => DELETED	3	5	0.6
	Generate Portfolio Instance - EX => DELETED			
	Add Comment - EX => NEW			
SUM				5.85
Instability				0.39
Stability				0.61

CHAPTER 10

CONCLUSION AND FUTURE WORK

In this chapter, we summarize our research and suggest some ideas for future work.

10.1 Conclusion and Thesis Contribution

The purpose of our research is to propose a suite of metrics that measures the stability of UML class diagrams, UML use case diagrams, and UML sequence diagrams. We performed a comprehensive survey on the proposed stability metrics, which shows that UML diagrams are not yet covered. The existing stability metrics target the source-code, and few of them have been validated theoretically.

The research methodology we followed to propose this suite of metrics starts with UML diagrams analysis. We identified all UML diagram elements, and selected a set of them to compute their unchanged values. The selection of these elements was based on two things; first, the elements that are not optional, and second, the elements must serve the meaning of the UML diagram. Then we selected an identifier in order to compare UML diagram versions. The identifier contains the minimum information that can be used to recognize the corresponding partner in the next UML diagram version so that we can make a correct comparison.

The UML diagrams are full of relationships; therefore, in order to avoid counting the changes more than once we proposed the Client Master approach. The Client Master approach is used to determine which side of the relationship is the client and which one is the master; the changes in

the relationship will be counted as being on the client side. Then we check all possible changes that may happen to any selected element in each UML diagram.

Finally we introduced our metrics suite to compute the unchanged properties in each UML diagram. These metrics are: the structural stability (SS) metric to measure UML class diagrams, the functional stability (FS) metric to measure UML use case diagrams, and the behavioral stability (BS) metric to measure UML sequence diagrams.

All metrics have been theoretically validated using the properties outlined by Kitchenham et al. We also applied our metrics on six different case studies, which are: Automated Teller Machine (ATM), Supply Chain Management (SCM), On Road Assistance (ORA), Online Real Estate Directory (O-RED), Hajj Online Services System (HOSS), and Electronic Students' Academic Portfolio (ESAP).

10.2 Future work

The following are some directions for future research:

- Provide a tool to compute the metrics suite. We need a tool that helps to perform the experiments easily and precisely.
- Empirically validate the proposed metrics, and correlate them with the maintenance process.
- In the research, we consider the elements that have been renamed as having been deleted, so we need to consider these elements without counting them as fully changed.
- UML sequence diagram fragments and constrains are not covered, so we need to extend the sequence diagram metric to consider them.

10.3 Threats to Validity

There are some threats that may affect the validity of the results. First, because there is no available tool to perform the experiment, we did it manually, which may have introduced errors as it is a human process. However, we verified the manual results more than once to overcome this threat. Another possible threat is that the experiments were done on small size projects; the use of large size projects may provide more confidence on the results.

Finally, the proposed metrics have not been empirically validated due to the lack of the UML diagrams' data, a correlation with the proposed metrics' values and defects may provide more confidence on the applicability of these proposed metrics. We plan to run such a validation once the data is available.

References

- [1] W. Li, "Software product metrics," *Potentials, IEEE*, vol. 18, pp. 24-27, 1999.
- [2] OMG, "OMG Unified Modeling Language™ (OMG UML), Superstructure," ed.
- [3] M. K. Daskalantonakis, "A practical view of software measurement and implementation experiences within Motorola," *Software Engineering, IEEE Transactions on*, vol. 18, pp. 998-1010, 1992.
- [4] M. Azuma, T. Komiyama, T. Miyake, S. Sakurai, A. Yamada, and T. Yonezawa, "Panel: the model and metrics for software quality evaluation report of the Japanese National Working Group," in *Computer Software and Applications Conference, 1990. COMPSAC 90. Proceedings., Fourteenth Annual International*, 1990, pp. 64-69.
- [5] OMG, "UML 2.0," *Object Management Group*, 2005.
- [6] M. Misbhauddin, "Towards An Integrated Metamodel Based Approach to Software Refactoring," Dissertation/Thesis, ProQuest, UMI Dissertations Publishing, 2012.
- [7] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard, "Object-oriented software engineering: a use case driven approach," 1992.
- [8] D. Pitone and N. Pitman, *UML 2.0: in a nutshell*. US: O'Reilly, 2005.
- [9] K. Sethi, Y. Cai, S. Wong, A. Garcia, and C. Sant'Anna, "From retrospect to prospect: Assessing modularity and stability from software architecture," in *Joint Working IEEE/IFIP Conference on Software Architecture, 2009 European Conference on Software Architecture. WICSA/ECSA 2009*, 2009, pp. 269-272.
- [10] A. Molesini, A. Garcia, C. von Flach Garcia Chavez, and T. V. Batista, "Stability assessment of aspect-oriented software architectures: A quantitative study," *Journal of Systems and Software*, vol. 83, pp. 711-722, 2010/05// 2010.
- [11] S. A. Tonus, A. Ashkan, and L. Tahvildari, "Evaluating architectural stability using a metric-based approach," in *Proceedings of the 10th European Conference on Software Maintenance and Reengineering, 2006. CSMR 2006*, 2006, pp. 10-pp.-270.
- [12] M. Jazayeri, "On Architectural Stability and Evolution," in *Reliable Software Technologies — Ada-Europe 2002*, J. Blieberger and A. Strohmeier, Eds., ed: Springer Berlin Heidelberg, 2002, pp. 13-23.
- [13] J. Bansiya, "Evaluating Framework Architecture Structural Stability," *ACM Comput. Surv.*, vol. 32, 2000/03// 2000.
- [14] H. Ma, W. Shao, L. Zhang, Z. Ma, and Y. Jiang, "Applying OO Metrics to Assess UML Meta-models," in *«UML» 2004 — The Unified Modeling Language. Modeling Languages and Applications*, T. Baar, A. Strohmeier, A. Moreira, and S. J. Mellor, Eds., ed: Springer Berlin Heidelberg, 2004, pp. 12-26.
- [15] M. Mattsson and J. Bosch, "Characterizing stability in evolving frameworks," in *Proceedings of Technology of Object-Oriented Languages and Systems, 1999*, 1999, pp. 118-130.
- [16] A. Moataz, R. Raimi, A. Jarallah, and K. Sohel, "Measuring architectural stability in object oriented software," *King Fahad University of Petroleum and Minerals, Dhahran*, 2003 2003.

- [17] Y. S. Hassan, "Measuring software architectural stability using retrospective analysis," M.S., King Fahd University of Petroleum and Minerals (Saudi Arabia), Saudi Arabia, 2007.
- [18] L. Aversano, M. Molfetta, and M. Tortorella, "Evaluating architecture stability of software projects," in *2013 20th Working Conference on Reverse Engineering (WCRE)*, 2013, pp. 417-424.
- [19] D. Grosser, H. A. Sahraoui, and P. Valtchev, "Predicting software stability using case-based reasoning," in *17th IEEE International Conference on Automated Software Engineering, 2002. Proceedings. ASE 2002*, 2002, pp. 295-298.
- [20] D. Grosser, H. A. Sahraoui, and P. Valtchev, "An analogy-based approach for predicting design stability of Java classes," in *Software Metrics Symposium, 2003. Proceedings. Ninth International*, 2003, pp. 252-262.
- [21] D. Rapu, S. Ducasse, T. Girba, and R. Marinescu, "Using history information to improve design flaws detection," in *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings*, 2004, pp. 223-232.
- [22] W. Li, L. Etzkorn, C. Davis, and J. Talburt, "An empirical study of object-oriented system evolution," *Information and Software Technology*, vol. 42, pp. 373-381, 2000/04/15/ 2000.
- [23] M. Alshayeb, M. Naji, M. O. Elish, and J. Al-Ghamdi, "Towards measuring object-oriented class stability," *IET Software*, vol. 5, pp. 415-424, 2011/08// 2011.
- [24] M. Alshayeb, "On the relationship of class stability and maintainability," *IET Software*, vol. 7, pp. 339-347, 2013/12// 2013.
- [25] M. O. Elish and D. Rine, "Indicators of Structural Stability of Object-Oriented Designs: A Case Study," in *Software Engineering Workshop, 2005. 29th Annual IEEE/NASA*, 2005, pp. 183-192.
- [26] M. Mattsson and J. Bosch, "Stability assessment of evolving industrial object-oriented frameworks," *Journal of Software Maintenance: Research and Practice*, vol. 12, pp. 79-102, 2000/03/01/ 2000.
- [27] M. O. Elish and D. Rine, "Investigation of metrics for object-oriented design logical stability," in *Seventh European Conference on Software Maintenance and Reengineering, 2003. Proceedings*, 2003, pp. 193-200.
- [28] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, pp. 476-493, 1994/06// 1994.
- [29] S. Raemaekers, A. van Deursen, and J. Visser, "Measuring software library stability through historical version analysis," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012, pp. 378-387.
- [30] D. Kelly, "A study of design characteristics in evolving software using stability as a criterion," *IEEE Transactions on Software Engineering*, vol. 32, pp. 315-329, 2006/05// 2006.
- [31] S. S. Yau and J. S. Collofello, "Some Stability Measures for Software Maintenance," *IEEE Transactions on Software Engineering*, vol. SE-6, pp. 545-552, 1980/11// 1980.
- [32] M. Alshayeb and W. Li, "An Empirical Study of System Design Instability Metric and Design Evolution in an Agile Software Process," *J. Syst. Softw.*, vol. 74, pp. 269-274, 2005/02// 2005.

- [33] H. M. Olague, L. H. Etzkorn, W. Li, and G. Cox, "Assessing design instability in iterative (agile) object-oriented projects," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, pp. 237-266, 2006/07/01/ 2006.
- [34] R. C. Martin and M. Martin, *Agile Principles, Patterns, and Practices in C#*.
- [35] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics," 1996.
- [36] J. F. Patenaude, E. Merlo, M. Dagenais, and B. Lague, "Extending software quality assessment techniques to Java systems," in *Seventh International Workshop on Program Comprehension, 1999. Proceedings, 1999*, pp. 49-56.
- [37] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein, "Pattern matching for clone and concept detection," *Automated Software Engineering*, vol. 3, pp. 77-108, 1996/06/01/ 1996.
- [38] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis, "Measuring clone based reengineering opportunities," in *Software Metrics Symposium, 1999. Proceedings. Sixth International, 1999*, pp. 292-303.
- [39] D. H. Qiu, H. Li, and J. L. Sun, "Measuring software similarity based on structure and property of class diagram," in *2013 Sixth International Conference on Advanced Computational Intelligence (ICACI), 2013*, pp. 75-80.
- [40] J. Krinke, "Identifying similar code with program dependence graphs," in *Eighth Working Conference on Reverse Engineering, 2001. Proceedings, 2001*, pp. 301-309.
- [41] C. Liu, C. Chen, J. Han, and P. S. Yu, "GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis," 2006, pp. 872-881.
- [42] J. H. Johnson, "Identifying Redundancy in Source Code Using Fingerprints," 1993, pp. 171-183.
- [43] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code," *IEEE Trans. Softw. Eng.*, vol. 32, pp. 176-192, 2006/03// 2006.
- [44] R. Hennicker and N. Koch, "Systematic design of Web applications with UML," in *Unified modeling language*, ed: IGI Publishing, 2001, pp. 1-20.
- [45] S. Berner, M. Glinz, and S. Joos, "A classification of stereotypes for object-oriented modeling languages," presented at the Proceedings of the 2nd international conference on The unified modeling language: beyond the standard, Fort Collins, CO, USA, 1999.
- [46] L. Briand, K. El Emam, and S. Morasca, "Theoretical and empirical validation of software product measures," *International Software Engineering Research Network, Technical Report ISERN-95-03*, 1995.
- [47] K. El-Emam, "A methodology for validating software product metrics," 2000.
- [48] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*: PWS Publishing Co., 1998.
- [49] K. Srinivasan and T. Devi, "Software Metrics Validation Methodologies in Software Engineering," ed: IJSEA, 2014.
- [50] L. C. Briand, J. W. Daly, and J. Wust, "A unified framework for cohesion measurement in object-oriented systems," in *Software Metrics Symposium, 1997. Proceedings., Fourth International, 1997*, pp. 43-53.
- [51] E. J. Weyuker, "Evaluating software complexity measures," *Software Engineering, IEEE Transactions on*, vol. 14, pp. 1357-1365, 1988.

- [52] B. Kitchenham, S. L. Pfleeger, and N. Fenton, "Towards a Framework for Software Measurement Validation," *IEEE Trans. Softw. Eng.*, vol. 21, pp. 929-944, 1995.
- [53] L. Maciaszek, C. González-Pérez, and S. Jablonski, *Evaluation of Novel Approaches to Software Engineering: 3rd and 4th International Conferences, ENASE 2008/2009, Funchal, Madeira, Portugal, May 4-7, 2008 / Milan, Italy, May 9-10, 2009. Revised Selected Papers* vol. 69. Berlin, Heidelberg: Springer, 2010.
- [54] L. C. Briand, Y. Labiche, and L. O'Sullivan, "Impact analysis and change management of UML models," in *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, 2003, pp. 256-265.
- [55] M. Chapman, M. Goodner, B. Lund, B. McKee, and R. Rekasius, "Supply Chain Management Sample Application Architecture," *Web Services Interoperability Organization*, 2003.
- [56] N. Koch, "Automotive case study: UML specification of on road assistance scenario," Technical Report 1, FAST2007.

Vitae

Name : Amjad Abu Hassan

Nationality : Palestine

Date of Birth :10/9/1987

Email : eng.abuhassan@gmail.com

Address : Yatta, Hebron, Palestine

Academic Background :Amjad Abu Hassan completed his Bachelors degree in computer systems engineering from Palestine Polytechnic University in January 2011. Since that he worked at Exalt technologies until he joined King Fahd University of Petroleum and Minerals in February 2013.