SOFTWARE SECURITY IMPROVEMENT THROUGH THE APPLICATION OF UML MODEL REFACTORING

BY

HARIS MUMTAZ

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

OCTOBER 2016

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN- 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by HARIS MUMTAZ under the direction of 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.

[illegible]

Dr. Adel Fadhl Ahmed
Department Chairman

M Alshayeb

Dr. Mohammad Alshayeb
(Advisor)

[illegible]

Dr. Salam A. Zummo
Dean of Graduate Studies

عمادة الدراسات العليا
DEANSHIP OF GRADUATE STUDIES

[illegible]

Dr. Mahmood Niazi
(Member)

6/11/16

Date

Sajjad Mahmood

Dr. Sajjad Mahmood
(Member)

DEDICATION

To my family,

especially my parents

ACKNOWLEDGMENTS

First of all, I would like to thank my thesis advisor Dr. Mohammed Alshayeb for his limitless support throughout the course of my research work. Whenever I faced a blind alley in my research, he guided me out and steered me to the right direction. The most important ingredient of the successful and timely completion of this thesis was his prompt feedback. His contributions were not just confined to the technical assessment but also assisted me in the writing of this thesis. His dedication has contributed significantly to the successful and timely completion of this thesis.

I would also like to thank my thesis committee members: Dr. Mahmood Niazi and Dr. Sajjad Mahmood. They have contributed significantly in the refinement of this thesis. Their critical evaluation has enhanced the quality of my work to a great degree. I am greatly indebted for their valuable comments on my thesis.

I would not do justice to my work if I do not acknowledge the incomparable support that I receive from my family. I must express my profound gratitude to my parents and my wife. They certainly have supported me with continuous encouragement throughout the course of my research. This accomplishment would not have been possible without them. I also would like to thank all of my friends especially Ahsan Javed, Abdullah Hejazi, Sajid Anwar, Taha Nasir, Abdullah Siddique and Mohsin Javed for the unforgettable memories.

Finally, I would like to thank Department of Information and Computer Science at King Fahd University of Petroleum and Minerals for providing me the opportunity to complete my master studies and Ministry of Education, Saudi Arabia for the generous funding.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

UML	:	Unified Modeling Language
GA	:	Genetic Algorithm
XMI	:	XML Metadata Interchange
XML	:	Extensible Markup Language
OCL	:	Object Constraint Language
CBO	:	Coupling Between Objects
RFC	:	Response For a Class
LCOM	:	Lack of Cohesion of Methods
WMC	:	Weighted Methods per Class
NOM	:	Number of Methods
DIT	:	Depth In Inheritance
NOC	:	Number Of Children
KLOC	:	Kilo Lines Of Code
NAttr	:	Number of Attributes
NAss	:	Number Of Associations
NInvoc	:	Number of Invocations
NRec	:	Number of Received messages
NOps	:	Number of operations
RPubAttr	:	Ratio of Public Attributes
RPriAttr	:	Ratio of Private Attributes
RProAttr	:	Ratio of Protected Attributes
RPriOps	:	Ratio of Private Operations
RProOps	:	Ratio of Protected Operations
EC	:	External Coupling
IC	:	Internal Coupling

ABSTRACT

Full Name	: Haris Mumtaz
Thesis Title	: Software Security Improvement through the Application of UML Model Refactoring
Major Field	: Software Engineering
Date of Degree	: October 2016

Software bad smells tend to have a negative impact on software quality by degrading a number of software quality attributes. It is imperative to detect and correct bad smells from analysis and design models to avoid their propagation to later stages of software development. Security is a vital quality attribute because of the critical nature of applications these days. In recent years, research related to secure software development has been observed as an uprising trend, however, there is a scarcity of corpus in investigating security bad smells and impact of refactoring on improving the software security. The main objective of this research is to vanquish the problem of security in UML models through the application of automated model refactoring. The fulfillment of the main objective is accomplished through multiple activities, which includes; proposing a detection technique, proposing a correction technique, empirical evaluation of proposed techniques and assessment of security improvement in UML models as a result of refactoring. The detection of security bad smells is achieved through the adaptation of a genetic algorithm, while correction is accomplished by model transformation approach. For the purpose of evaluation, our study focuses on three UML models (use case diagram, sequence diagram and class diagram). The assessment of security improvement is accomplished through statistical analysis of quality metrics. The empirical validations of proposed approaches are performed through multiple case studies of investigated UML models. The results show significant detection recall and correction efficacy of our proposed detection and correction approaches respectively. Besides automatic detection and correction, the identification and refactoring of security bad smells are validated manually as well. The manual assessment of investigated models; and statistical analyses of quality metrics allow us to conclude the significant improvement in security quality of investigated UML models as a result of refactoring.

ملخص الرسالة

الاسم الكامل: حارث ممتاز

عنوان الرسالة: تحسين أمن البرمجيات من خلال تطبيق إعادة هيكلة البرمجيات لنماذج لغة النمذجة الموحدة

التخصص: هندسة البرمجيات

تاريخ الدرجة العلمية: أكتوبر 2016

للروائح الكريهة للبرمجيات تأثير سلبي على جودة البرمجيات عن طريق تأثيرها على عناصر جودة البرمجيات. لا بد من كشف وتصيحي الروائح الكريهة للبرمجيات في نماذج التحليل والتصميم لتجنب انتشارها إلى مراحل لاحقة من تطوير البرمجيات. امن البرمجيات سمة مهمة في جودة البرمجيات بسبب الطبيعة الحرجة للتطبيقات في هذه الأيام. في السنوات الأخيرة، لوحظ تزايد البحوث ذات الصلة لضمان تطوير البرمجيات، ومع ذلك، هناك عدد قلقل من البحوث التي تعالج تأثير الروائح الكريهة للبرمجيات على امن البرمجيات. الهدف الرئيسي من هذا البحث هو إيجاد حل لمشكلة الأمن في نماذج لغة النمذجة الموحدة من خلال تطبيق إعادة هيكلية البرمجيات بشكل آلي. ويتم تحقيق هذا الهدف الرئيسي من خلال أنشطة متعددة، والتي تشمل؛ اقتراح تقنية لكشف عن الروائح الكريهة للبرمجيات، واقتراح طريقة لتصحيح الروائح الكريهة للبرمجيات والتقييم التطبيقي لهذه الطرق المقترحة وتقييم التحسن الأمني في نماذج لغة النمذجة الموحدة نتيجة لإعادة هيكلة البرمجيات. ويتحقق الكشف عن الروائح الأمنية السيئة للبرمجيات من خلال تطويع الخوارزمية الجينية، بينما يتم إنجاز التصحيح من قبل نهج نموذج التحول. لغرض التقييم تركز دراستنا على ثلاثة نماذج من لغة النمذجة الموحدة وهي حالة الاستخدام، وتسلسل الرسم وفئة الرسم. ويتم إنجاز تقييم التحسن الأمني من خلال التحليل الإحصائي لمقاييس الجودة. يتم تنفيذ عمليات التحقق التجريبية للنهج المقترح من خلال دراسات لنماذج ممثلة بلغة النمذجة الموحدة. أظهرت النتائج قدرة عالية على الكشف وتصحيح الروائح الكريهة للبرمجيات. إلى جانب الكشف والتصحيح التلقائي، تم التأكد من إعادة الهيكلة للروائح الأمنية السيئة يدويا كذلك. التقييم والتحليلات الإحصائية لمقاييس الجودة تأكد لنا وجود تحسن كبير في جودة أمن نماذج لغة النمذجة الموحدة لتحقيق إعادة هيكلة البرمجيات.

CHAPTER 1

INTRODUCTION

Unified Modeling Language (UML) is a widely used analysis and design language because of its supportability towards a number of software quality attributes [1]. It allows the designers to develop analysis and design models ensuring important quality attributes. A number of quality attributes related to software modeling have been reported in the literature, such as modularity, reusability, modifiability, testability, security etc. [2]. Software models (such as use case diagrams, sequence diagrams and class diagrams etc.) have been rigorously analyzed by researchers to ensure the presence of these quality attributes. The quality attributes may suffer if poor analysis and design decisions are taken during software development. The poor design and implementation decisions are commonly referred as ‘bad smells’, and necessary measure taken to remove the bad smells is called ‘refactoring’ [3].

The bad smells are usually categorized as code and model bad smells. Model bad smells mainly focus on analysis and design defects, which may hinder in later stages of software development. On the other hand, code bad smells are only confined to common inappropriate implementation practices [4]. The most beneficial extraction from the studies of bad smells and related refactoring strategies is the improvement of software quality. The end objective of each study in the context of refactoring is to enhance the commonly reported quality attributes. A number of studies have developed automated detection and correction techniques and tools for code bad smells [5]. A comparatively fewer number of studies have worked on automated detection of bad smells and related refactoring opportunities for software models [6].

1.1. Problem Statement

A decent corpus has focused on improvement of a variety of software quality attributes through model refactoring [7-9]. However, there is a scarcity of literature on the impact of model refactoring on security quality of a software. Security has a significant importance because of the nature of applications these days. Secure software development is a widely researched domain and it seems to have its pace enhanced in recent years [10, 11]. Detection of security bad smells becomes an essential task in this regard. Finding security related smells in models is not sufficient as it does not fully fortify the problem. For eradication of security smells, recognition of appropriate refactoring techniques must be obliged. To the best of our knowledge, refactoring opportunities suggested in the literature, unfortunately, do not focus on security aspects from both software analysis and design perspectives [11-13].

1.2. Motivation

A rigorous literature review on the studies related to model smells, detection strategies and refactoring techniques has allowed us to identify few gaps, which motivate us to work in this area. The related literature has not yet studied the model smells from a security point of view. The literature studies mostly focus on proposing more bad smells or studying the impact of existing bad smells on the quality of source code and UML models. The literature studies are also leaned towards detection and refactoring of bad smells in class diagrams.

Another concern to be noticed is the lack of work on use case diagram and sequence diagram in the context of refactoring. Few studies address the issues related to refactoring in use case diagram and sequence diagram [14-16]. The researchers have not yet addressed detection and refactoring of bad smells from a security perspective. These significant gaps in the literature stimulate our motivation to study bad smells from a security perspective and include other models such as use case diagram and sequence diagram as well. To the best of our knowledge, no study exists whose objective is to provide automated detection and refactoring approaches for security bad smells in software models. This further motivates us to investigate model security bad smells and propose automated detection and refactoring techniques to identify and eradicate security bad smells from software models.

1.3. Research Objectives and Questions

The main goal of this research is to improve the security of software models through the application of refactoring. The achievement of this goal can be broken down into multiple sub-objectives. The sub-objectives of this research include:

- Propose a detection technique to identify security bad smells in UML models.
- Propose a correction technique to eradicate security bad smells in UML models.
- Empirical assessment of security improvements in UML models as a result of refactoring.

This research aims to cover all three views of UML. One model is selected from each UML aspect i.e. class diagram from structural; sequence diagram from behavioral; and use case diagram from functional [1]. Our research aims to address the following laid research questions:

RQ1: To what extent can our proposed detection approach detect security bad smells in UML models?

RQ2: To what extent can our proposed correction approach rectify security bad smells in UML models?

RQ3: To what extent can refactor to security bad smells improve security aspects of UML models?

1.4. Research Methodology

In order to address RQ1 and RQ2 specified in the previous section, we propose detection and correction approaches respectively. The detection approach uses the concept of Genetic Algorithm (GA) to identify security bad smells in studied UML models. A potential solution is formed by creating a set of rules measuring for security bad smells using quality metrics. The approach does not require any manual expression of detection rules because they are based on existing security bad smells examples. The use of examples also does not require specification of quality metrics thresholds. The best solution is yielded through selection, crossover and mutation operations of GA process. The correction solution is based on model transformation using XMI. The XML representation of a corresponding model is refactored to remove security bad smells. The refactored XML is then exported to the corresponding UML model. RQ3 is answered through manual analysis of investigated case studies of UML models and statistical analysis of software quality metrics. The comparison of software metrics values before and after refactoring allow a definite conclusion on significant security improvement in software models.

1.5. Research Contribution

The major contributions to the research and professional community are listed below:

- Taxonomy of security bad smells.
- Catalog of refactoring to security for analysis and design models (class, use case and sequence diagrams).
- A method for the automatic detection of security bad smells for analysis and design models (class, use case and sequence diagrams).
- A method for the automatic refactoring to security for the analysis and design models (class, use case and sequence diagrams).
- Empirical evaluation of security improvements in analysis and design models (class, use case and sequence diagrams).

1.6. Thesis Outline

The rest of this thesis is structured as follows: chapter 2 provides preliminary background on some key concepts. It presents illustration on UML, software models security attributes, metrics and refactoring. Chapter 3 provides a detailed description of related work. Chapter 4 encompasses our research methodology. Chapter 5 encapsulates the application of the proposed detection and correction approaches on multiple case studies of considered UML models. It also describes the validation of proposed approaches through case studies. Since this chapter provides explanations of our experiments, it follows the guidelines provided by Jedlitschka et.al on reporting empirical studies in software engineering [17]. Chapter 6 analyses and discusses the implications of our acquired results. Chapter 7 presents posed threats to validity and finally, Chapter 8 concludes the thesis and directs future work.

CHAPTER 2

BACKGROUND

This chapter provides a preliminary background on the important concepts and aspects expected in our research domain. Following sections shed light on UML, security attributes in software design, software metrics and refactoring. The focus of our research is the class diagram, sequence diagram and use case diagram of UML, so the explanation of each model is provided accordingly.

2.1. Unified Modeling Language (UML)

The Object Oriented paradigm has collected popularity since the last two decades because of its conceptual modeling nature. UML was introduced to provide software modeling standard [1]. The benefit of developing a standard language was to bring developers from all over the world to a single software modeling platform. UML uses graphical notations to design software systems [18]. The language keeps on evolving since the time it is proposed and currently, UML 2.0 is in use for software modeling [19]. UML can be viewed from three different perspectives: structural, behavioral and functional [1]. The models lie in each classification are as follows:

Structural View: Class diagram, object diagram, package diagram, deployment diagram, composite structure diagram and component diagram.

Behavioral View: Sequence diagram, communication diagram, timing diagram and interaction view diagram.

Functional View: Activity diagram, use case diagram and state machine diagram.

It can be noticed that each view has many models and covering all the models is beyond the scope of our research. Our work covers all three views by selecting one model from each view. Since class diagram, sequence diagram and use case diagram are the most widely applied models for software modeling [6, 10], our thesis research is confined to these three models. A brief description of each model is provided underneath.

Class Diagram: Class diagram provides a static structure of objects sharing attributes and procedures. It depicts a conceptual design, which later, is translated to implementation. The class diagram also shows relationships with other classes. The relationships are usually association, aggregation and generalization. The example, in Figure 1, illustrates the basic components of class diagram. The example presents a structural view of online purchasing of goods. Although the presented example is self-explanatory but to further ease the understanding, brief illustration is provided. 'Customer' and 'Item' are distinctive entities formulating the structure of class diagram, so they are represented as classes. The 'Item' class has two attributes: 'shippingWeight' and 'description', and two procedures: 'getPriceforQuantity' and 'getWeight'. 'Customer' and 'Order' classes show association relationship as well and it can be noticed that it can be bidirectional or unidirectional. The association relationship also caters with a cardinality of relationship, for example, an association relationship can be one to one; one to many; or many to many, depending on the relationship between classes. The classes 'Order' and 'OrderDetail' share a relationship of aggregation. The aggregation relationship is understandable by its literal meaning. In the example, one or many order details will aggregate to a single order. The customer can make payment in multiple ways i.e. via check, cash and credit card. These three payment modes can be generalized to make an abstract class.

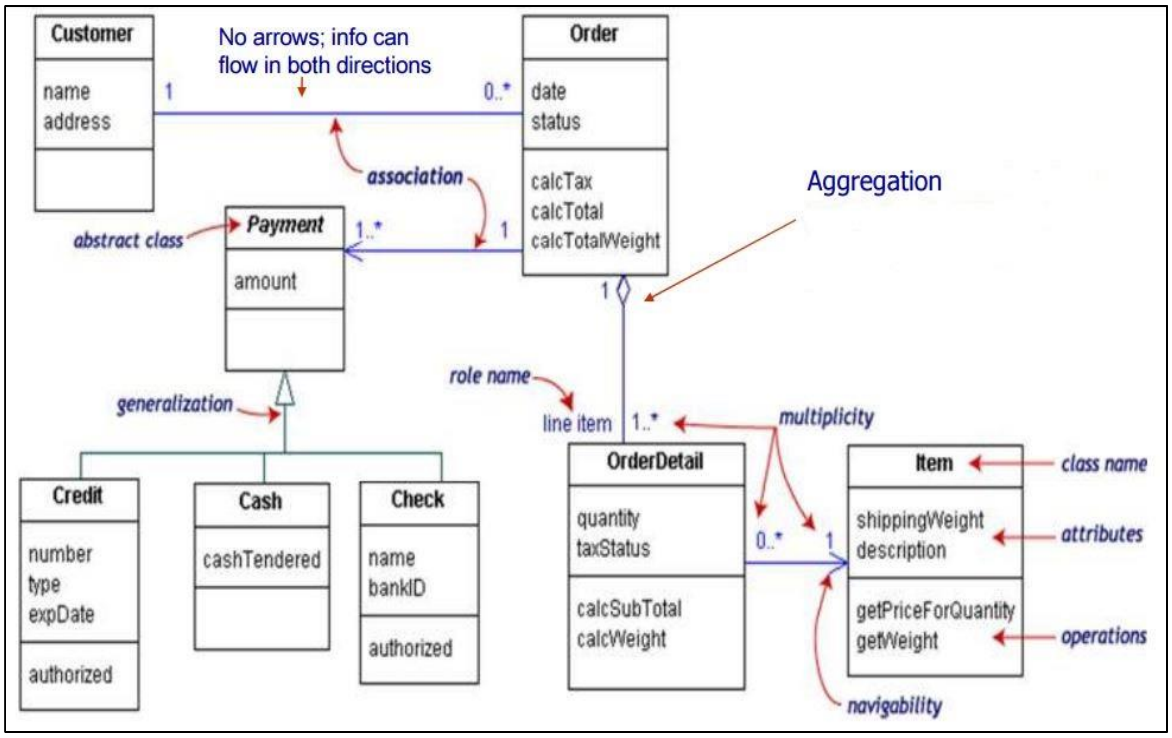

Figure 1. Class diagram overview [20]

Sequence Diagram: UML sequence diagram models the flow of logic within a system in a visual manner, and is commonly used for both design and analysis purposes [21]. It normally represents the series of messages sent to and fro objects over time. The main purpose of a sequence diagram is to represent usage scenarios and explore the complex operational and procedural logics [21]. A small example, in Figure 2, illustrates working of the sequence diagram. In the figure, ‘Student’ is representing a class with ‘aStudent’ an instance of it. The instances of ‘Seminar’ and ‘Course’ classes are kept anonymous because they do not need to be referenced in the diagram. The ‘Student’ is sending messages to ‘Seminar’ by making function calls, similarly, ‘Seminar’ and ‘Course’ classes. The dashed lines known as object lifelines represent the life span of the object over time. The thin boxes on objects lifelines are activation boxes, representing the object's active time during a communication with other classes [21].

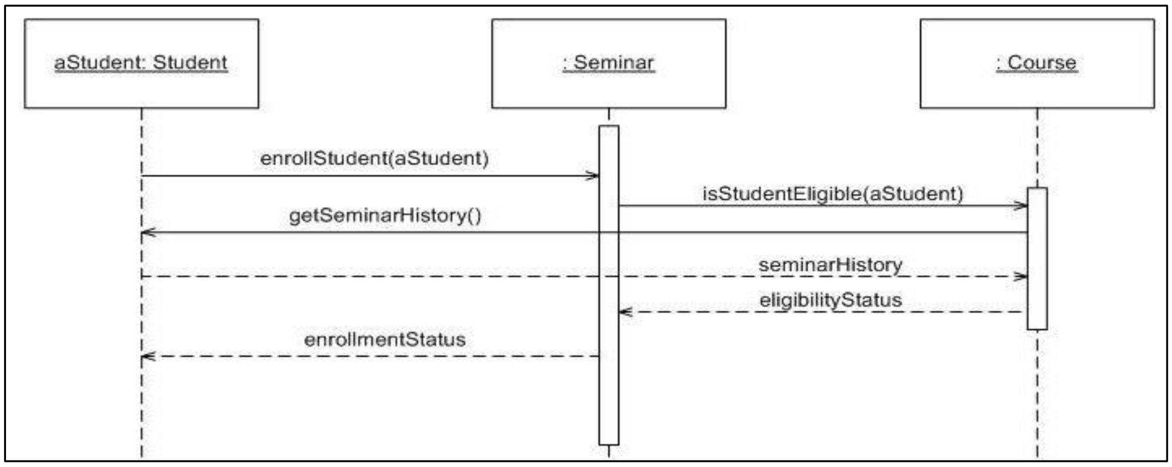

Figure 2. Sequence diagram overview [21]

Use Case Diagram: The primary objective of use case modeling is to elicit functional requirements of a system. A use case diagram provides a graphical representation of how actors interact with the system [22]. It has entities like actors, use cases and relationships. An actor accomplishes a service through a use case. The construction of use case models can be done in multiple ways i.e. informal, semi-structured, or fully structured [23]. Use case diagram, in Figure 3, shows a functional view of hospital's reception. The 'Receptionist' is represented as an actor because he/she is supposed to interact with the system and accomplish certain duties. He/she performs many tasks including scheduling appointments, admission to hospital, collecting patient's information etc. Some tasks can only be accomplished by performing sub-tasks, for example, 'InPatient Hospital Admission' includes 'Bed Allotment'. The extends relationship is an optional task performed as an extension to another task.

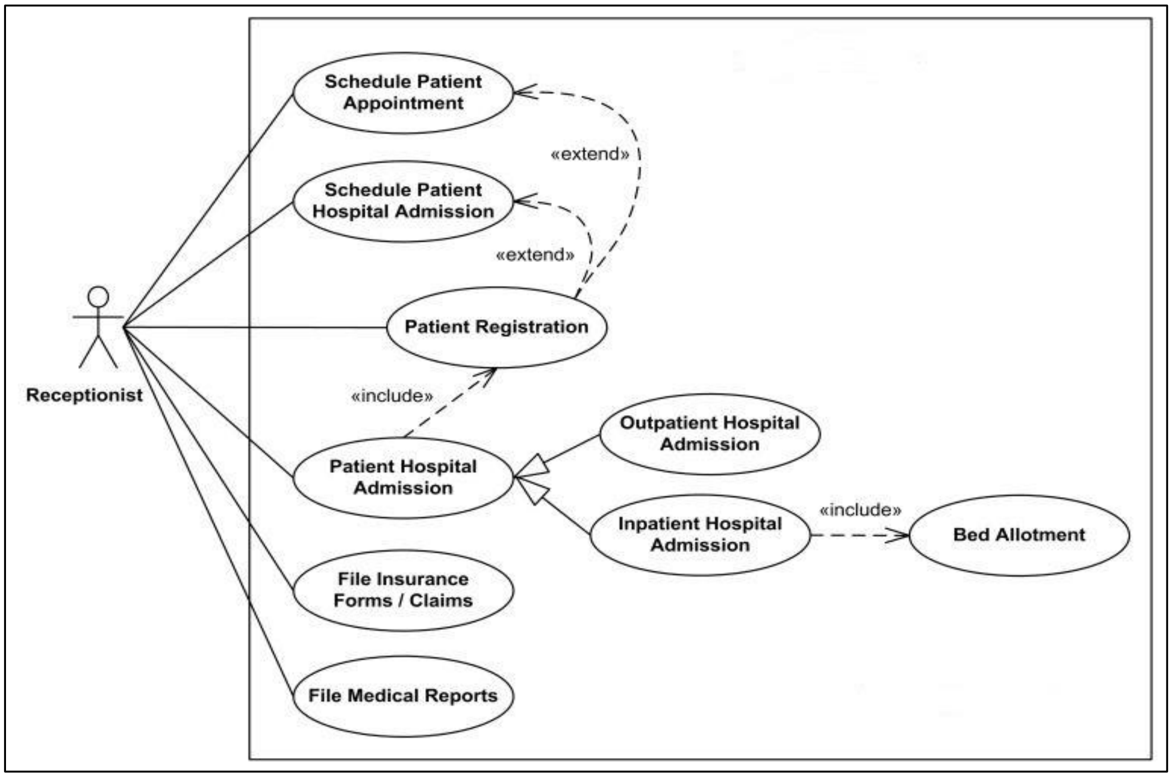
Schedule Patient Appointment
Schedule Patient Hospital Admission
«extend»
«extend»
Patient Registration
Receptionist
«include»
Patient Hospital Admission
Outpatient Hospital Admission
Inpatient Hospital Admission
«include»
Bed Allotment
File Insurance Forms / Claims
File Medical Reports

Figure 3. Use case diagram overview [22]

2.2. Security Attributes

A number of software quality attributes have been reported in literature such as performance, scalability, modifiability, security, availability, integration, portability and testability [2]. Many researchers have studied the measurement of quality attributes in object-oriented code using different techniques [24-26]. Object-oriented metrics is a renowned method for measuring software quality attributes. Security is one of the important quality characteristics in software models. The most commonly reported security attributes are: confidentiality, integrity and availability [2, 10, 27, 28].

According to Whitman, information security is to protect the information in terms of its confidentiality, integrity and availability in all means: storage, processing and transmission [28]. Confidentiality, integrity and availability are major ingredients for ensuring security.

When information is protected from unauthorized access, it means confidentiality is ensured [28]. The integrity of information is compromised when information is exposed to damage, corruption or any kind of disruption [28]. The violation of confidentiality and integrity leads to critical problems such as reliability, consistency, completeness and correctness [29]. The third major security attribute is availability, which means that data and services are available to authorized users at all the times [28].

Jürgen listed some important security characteristics including fair exchange, non-repudiation, role-based access control, secure communication link, secrecy and integrity, authenticity, freshness, secure information flow, guarded access [10]. Gorton pointed some security requirements that a software system should encapsulate. The security requirements include authentication, authorization, encryption, integrity and non-repudiation [2].

2.3. Software Metrics

Software metrics provide a quantitative measurement of different software artifacts [30]. The common use of software metrics mainly lies in software design and implementation. Several metrics have been proposed in software engineering. The reported object-oriented metrics are majorly segregated into four groups: coupling, cohesion, complexity and inheritance. The brief illustrations of each group and related metrics are presented below:

Coupling: It measures the interdependency between classes or objects. The inter-linking can be in the form of variable usage or method usage. The typical metrics used for measuring coupling are CBO (Coupling Between Objects) and RFC (Response For a Class). CBO is the count of number of classes that a class is coupled with [31]. It is desirable to have CBO value to be minimum because higher value leads to maintenance issues. RFC is the count for number of methods called by each method in a class and set of methods defined by that class [31]. The greater the number of methods, the more the RFC value.

Cohesion: It deals with the concept of separation of concerns for a class. It measures how strongly cohesive the components of a class are. It enforces the use of local attributes by local methods. LCOM (Lack of Cohesion of Methods) is a commonly used metric for measuring cohesion. It counts the number of method pairs whose similarity is zero, minus the count of method pairs whose similarity is not zero [31].

Complexity: It measures the simplicity of a design. There are many metrics reported in the literature to measure complexity. WMC (Weighted Method per Class) computes complexity by taking the summation of local complexity of each method [31]. NAtt (Number of Attributes) measures complexity by counting total number of attributes of a class [31]. The greater the number of attributes, the more the complexity. Lastly, NOM (Number Of Methods) calculates complexity by counting total number of methods of a class [31]. Similar to NAtt, the more the NOM, the more complex a design would be.

Inheritance: It is measured by two metrics DIT (Depth of Inheritance) and NOC (Number Of Children). DIT is the depth in the inheritance tree and NOC is the number of children of a class [31]. The greater the counts for these two metrics, the more difficult it is to maintain a design.

The use of software metrics in the context of refactoring is extremely imperative as they assist in evaluating the quality improvement as result of refactoring. Although quality assessment is an imperative activity in the refactoring process but few literature studies focused on it [6]. Enckevort used Fan-in and Fan-out in addition to C&K metrics to quantify model quality [32]. They used metrics values pre and post refactoring to analyze the impact of them on software quality. Moghadam and Cinnéide made use of cohesion metrics proposed by Al Dallal and Briand [33, 34]. The cohesion metrics belonged to method cohesion and class cohesion. Jensen and Cheng applied QMOOD metric suite to analyze the quality of software as a result of refactoring. QMOOD suite consists of 11 metrics and is proposed by Bansiya and Davis [35].

2.4. Refactoring

Software refactoring means that software design or code is transformed in such a way that it improves software quality while preserving its behavior [3]. Opdyke introduced the concept of software refactoring and proposed design and implementation level refactoring [36]. The list is compiled in Appendix B of this thesis.

2.4.1. Refactoring Process

Refactoring process is conducted in a series of steps. Refactoring for both code and model can be executed in an almost similar fashion. The whole process is suggested by Wake and William [4], and later extended by Mens et.al [37]. Mainly refactoring process includes following steps:

Step 1: Identification of part of software that requires refactoring

Step 2: Selection of appropriate refactoring

Step 3: Check for behavior preservation

Step 4: Apply refactoring

Step 5: Analyze the effect of refactoring in terms of improvement of quality.

Step 6: Ensure consistency between refactored different software artifacts.

The details related to step 1 and 2 are provided in related work. The explanations for detection techniques provide a detailed discussion on step 1 and 2. The third step involves code or model behavioral preservation, whose goal is to resist any changes, which may lead to a change in behavior. This step of refactoring process is usually achieved by defining pre and post conditions of refactoring [36, 38]. Opdyke used the assistance of preconditions to ensure behavioral preservation before refactoring [36]. Preconditions allow modelers to ensure that output result will be same regardless of refactoring. The only seeable drawback of imposing preconditions is its additional overhead to the refactoring process. Roberts et.al extended the refactoring procedure, to verify behavioral preservation, with post-conditions [38]. The rationale of using post-conditions instead of precondition was the belief of delaying the refactoring process, because verifying behavioral preservation before refactoring will delay the actual application of refactoring. Delaying the verification goal after refactoring application would also allow modelers to evaluate the effectiveness of refactoring techniques in the eradication of bad smells.

Post refactoring application leads to step 5, where quality assessment is performed. The goal of refactoring is to improve the quality. The assessment is usually accomplished through the use of quality metrics because of their objective nature. Utilizing metrics to evaluate quality is not sufficient, proposing appropriate models to effectively apply the metrics are equally imperative. In this regard, few studies presented quality models that use software metrics to make quality assessments. Lange and Chaudron proposed a model to assess quality change in software as a result of refactoring [39]. Their model uses C&K metrics to judge the quality change. Jalbani et.al provided a quality engineering methodology for UML models [40]. Their approach consisted of two basic parts: quality assessment and improvement. The quality model constructed through quality assessment included metrics proposed by Lange and Chaudron [39]. The second part of quality improvement focuses on model smell detection and refactoring. Yue et.al also presented a quality measurement approach that defines quality metrics at the meta-model level [41].

The last step in refactoring process deals with consistency issues in refactored artifact and other software artifacts. For example, if bad smells are identified in code and dealt with refactoring, the relevant modifications in the class diagram should also be accompanied. The dependency among software development phases urges the need to consider consistency while refactoring any artifact. Spanoudakis and Zisman categorized consistency into two types: vertical consistency and horizontal consistency [42]. Vertical consistency is concerned with changes in a single model, while, horizontal consistency deals with ensuring consistency between different UML models. Massoni presented some approaches to handling code and model consistency [43]. The three proposed techniques are; Simple forward engineering, Successive reverse engineering and Round-trip engineering. Simple forward engineering happens when models are usually discarded once implementation stage finishes so source code modifications would not create consistency issues [43]. The use of reverse engineering from source to the model can overcome inconsistency issues [43]. Once implementation stage is finalized and stable, the model can be refactored accordingly, and is called round-trip engineering [43].

Since the consistency mechanisms rely on restructuring model according to code modifications, Bottoni et.al proposed a reverse engineering approach based on coordinated graph transformation scheme [44]. Type graphs and flow graph are used to represent model and code respectively, while interface graph represents common interface parts between model and code. Common interface parts are represented using an interface graph. Refactoring is applied based on common interface changes. This manner, modification made at code level are reflected at the model level. However, the applicability of this approach is confined to an abstract level, meaning, it cannot be applied to changes made in method bodies. To cater this concern, Van Gorp et.al presented source consistent refactoring [45]. They provided an extended meta-model called GrammyUML [45]. GrammyUML allows modelers to deal with method level details. The studies highlighted so far in this paragraph focus on vertical consistency. The importance of horizontal consistency cannot be overlooked. Bottoni et.al alongside vertical consistency, applied the concept of coordinated graph transformation to horizontal consistency [44]. Tsiolakis coped with horizontal consistency by the application of attributed graph grammar [46]. They evaluated their approach on class diagrams and sequence diagrams.

2.4.2. Refactoring Tools

Application of the refactoring to source code and models can be achieved in a fully-automated manner, semi-automated manner or manually. Many refactoring tools have been developed for code refactoring. Table 1 lists down few refactoring tools alongside the targeted bad smells. Some other refactoring tools, targeting only models smells, are also found in the literature [47-49].

Table 1. Refactoring tools

Sr. No.	Tool	Targeted Bad Smells
1	InterlliJ IDEA [50]	Rename and Move Program Entities, Change Method Signature, Extract Method, Inline Method, Introduce Variable, Introduce Field, Inline Local Variable, Extract Interface, Extract Superclass, Encapsulate Fields, Pull Up Members, Push Down Members and Replace Inheritance with Delegation.
2	RefactorIt [51]	Rename, Move Class, Move Method, Encapsulate Field, Create Factory Method, Extract Method, Extract Superclass/Interface, Minimize Access Rights, Clean Imports, Create Constructor and Pull Up/Push Down Members.
3	JRefactory [52]	Move Class, Rename Class, Add an Abstract Superclass, Remove Class, Push Up Field, Pull Down Field and Move Method.
4	jFactor [53]	Extract Method, Rename Method Variables, Introduce Explaining Variable, Inline Temp, Inline Method, Rename Method, Pull Up/Push Down Method, Rename Field, Pull Up/Push Down Field, Encapsulate Field and Extract Superclass/Interface.

2.4.3. Commonly Applied Refactoring Strategies

Many refactoring strategies have been reported in the literature, but those applied in this research are briefly explained in this section. However, the comprehensive list of all refactoring strategies is provided in Appendix A. The refactoring strategies are collected from multiple resources [3, 29, 54, 55]. They are categorized as code, models or both. The categorization is based on the applicability of refactoring strategies at design and implementation levels. The refactoring strategies for the class diagram and sequence diagram are generally the same, while use case diagram has their dedicated refactoring strategies. The rationale behind is the use of classes and their interactions in class and sequence diagram. On the other hand, use case diagram expresses functional requirements using use cases and actors. The refactoring strategies for use case diagram are discussed in section 3.2. The refactoring strategies considered in this research, for class and sequence diagrams, are briefly explained below:

Move method: This means moving a method from a class to another class that uses this method more. That method in the class can be turned into a delegation or can be completely removed. The corresponding bad smell for this type of refactoring is ‘broken modularization’ i.e. a method uses more features of another class than the class it belongs to. This bad smell exploits the common object oriented design principle of modularization. Although modularization is introduced by distributing the methods across multiple classes but the separation of concerns is not ensured. The move method refactoring allows the method to move to the class where it is mostly required. This way modularization and separation of concerns are complemented.

Extract Class: This means moving cohesive methods and related attributes from an existing class to a new class. This type of refactoring handles the modularization of a design. This type of refactoring copes with the bad smell of missing modularization. This bad smell also violates the separation of concerns principle in software design. In other words, a class is overburdened by many responsibilities. This way high coupling is also induced. In order to cope with all these design principles violations, extract class refactoring is applied. The refactoring readjusts the design with the objective of having better modularization, separation of concerns and coupling.

Remove class: This means removing a class which is contributing nothing to a design. Such a class becomes useless and it is pointless to show it in the design. The idle class may incur inappropriate behavior in the case where it is accidently invoked. So, removing such class also increases the reliability of a design.

Encapsulate class: This means converting the access modifiers of a class from public to private. These public attributes allow the outside classes to obtain unguarded access to them. The unauthorized access can be restricted by properly encapsulating the class. The unauthorized access leads to inappropriate use of data by other classes, which contradicts with the concept of data confidentiality, integrity and reliability. Encapsulate class refactoring restricts these violations and enforces other class to communicate with the encapsulated class appropriately. The communication is achieved through getter and setter methods.

CHAPTER 3

RELATED WORK

This chapter provides a detailed discussion on studies related to our research. The illustration is presented in separate sections to ease the understanding and readability. This chapter covers the bad smells (code and model), security aspects of software development and existing bad smells detection techniques and tools.

3.1. Code Bad Smells

Fowler et.al suggested that code bad smells give a good indication of code issues and can be resolved by refactoring them [3]. They initially introduced 22 code smells and the number keeps on increasing since then. A comprehensive list of bad smells is available online [54]. According to Fowler et.al, Duplicated Code is widely investigated and frequently occurred bad smell in source codes [3]. The other widely studied bad smells are Feature Envy, Refused Bequest, Data Class, Long Method and Large Class [5]. The rest from the initial 22 bad smells are investigated rarely [5].

They initially referred listed smells as code bad smells but some of the listed bad smells are applicable to models as well. For example, 'Shotgun Surgery' is one of the presented bad smells and can be seen as both code bad smell and model bad smell [3]. Shotgun surgery occurs when classes are too much coupled, making a change in one class creates a ripple effect for other dependent classes. The appropriate refactoring suggested for this bad smell is to reduce the coupling between classes. Some studies aim to propose methods to detect code bad smells [37, 56]. Munro used software metrics to identify code bad smells [56], whereas, Mens and Tourwé applied logic meta programming to find code bad smells [37]. Cushman and Rosenberg suggested that objective methods should be used in combination with subjective methods because some attributes cannot be quantified and subjective methods can help in capturing software properties [57]. So the studies incorporating software metrics can extend the approaches to capture attributes through subjective methods.

Few studies investigated the impact of bad smells on code and models. Shatnawi and Li investigated the correlation of bad smells with software faults and their severity levels [58]. They found large class, large method and shotgun surgery to be significantly correlated with software faults. Their results indicated some bad smells like, data class, feature envy and refused bequest are not significantly associated with software faults. Monden et.al studied the effect of duplicated code on software reliability and maintainability [59]. They found that duplicated code reduces maintainability and reliability of a software. Kapser and Godfrey aimed to find different patterns via which duplicated code can be identified [60]. They identified 11 different duplicated code patterns. Domain experts were invited to judge the harmfulness of each pattern. The experts suggested that not all duplicated codes are harmful and would not require refactoring. One study targeted bad smells in an order to reduce the risk and improve the effectiveness of refactoring [61]. To obtain a further deep insight, there are few well-reported literature surveys on code and design bad smells [5, 37, 62].

3.2. Model Bad Smells

At the design level, many code smells can be traced back to the class diagram, thus can be referred as model smells. For example, in a data class, if data members are public, they are exposed to other classes. These public data members can be seen in the class diagram. Suryanarayana et.al classified design smells into four main categories: Abstraction, Encapsulation, Modularization and Hierarchy [29]. The classification with corresponding design smells is listed in Table 2. In each classification, a number of design smells are reported. For example, in abstraction, there is a design smell, 'missing abstraction', which emphasis on a compromise on the integrity of data. Similarly, in 'deficient encapsulation', the attributes of a class are likely to be exposed to outside classes. They also suggested some appropriate set of refactoring opportunities for each design smells and also their impact on quality attributes.

Table 2. Classification of design smells [29]

Classification	Design Smells
Abstraction	Missing, Imperative, Incomplete, Multifaceted, Unnecessary, Unutilized, Duplicate
Encapsulation	Deficient, Leaky, Missing, Unexploited
Modularization	Broken, Insufficient, Cyclically Dependent, Hub-like
Hierarchy	Missing, unnecessary, Unfactored, Wide, Speculative, Deep, Rebellious, Broken, Multipath, Cyclic

The design bad smells that are considered in this research are briefly described below:

Missing Hierarchy: A hierarchy should have been created to avoid unexpected hierarchical behavior and encapsulate expected variations. This design smell can be removed by creating a connection with appropriate hierarchy interface.

Missing Modularization: This type of design smell arises when a class or component is not decomposed. In other words, the component lacks in the separation of concerns. The appropriate refactoring strategy for removing this design smell is ‘Extract Class’. This way the cohesive attributes and methods are moved to a new class, which leaves old class properly modularized.

Broken Modularization: This bad smell happens when the data and related procedures are split across abstractions. This allows unauthorized access of data across classes or components. The related refactoring to eradicate this design smell is move method(s) and attribute(s). This way the data and related procedures are moved to the class(es) or components, where they actually belong.

Unutilized Abstraction: This design smell occurs when an unused abstraction is accidentally invoked. It may result in runtime problems, affecting the reliability of a design. This type of design smell can be erased by the application of remove abstraction refactoring. Removal of unutilized abstraction motivates correct invocation of objects, which result in the reliable execution of a software.

Deficient Encapsulation: This design smell provides direct access of class’s data to outside classes, compromising the confidentiality and integrity of a design. This type of design smell is extremely critical for data classes. The viable refactoring for this smell is to properly encapsulate the class.

Bad smells in the sequence diagram are studied in the context of abovementioned class diagram bad smells. Bad smells which belong to class diagram are applicable to a sequence diagram. For example, broken modularization is one of the bad smells usually experienced in the class diagram, can also be applied to a sequence diagram. The calling of methods between classes identifies how much classes are delved into each other, intimating the presence of broken modularization. The rectification procedure is also similar to the class diagram. If the bad smells are removed at the class level, they are automatically removed from the sequence diagram. Following the same broken modularization example, the methods are moved to remove the bad smells in classes, meaning, less calling of methods, eventually, fewer calls are observed in corresponding sequence diagrams.

Few studies have addressed the issue of bad smells in use case diagram and its description [14-16, 63]. Generally, defects in use cases are referred to as anti-patterns instead of bad smells. In our research, we are treating anti-patterns in use cases as bad smells. The major contribution to study anti-patterns in use case diagram is provided by El-Attar and Miller [14-16]. Their objective was to improve the quality of use case diagram and its description. They utilized anti-patterns to identify defects in use cases and refactor them to improve the quality of use cases. Their study influenced use cases quality in terms of correctness, consistency, analytical ability and understandability. Table 3 depicts the investigated use cases anti-patterns and their respective refactoring strategies.

Table 3. Use cases anti-patterns and corresponding refactoring techniques [14-16]

Anti-pattern	**Refactoring**
Accessing a generalized concrete use case	Concrete to Abstract, Drop Actor-Generalized UC Association
Accessing an extension use case [missing hierarchy]	Drop Actor-Extension UC Association
Using extension/inclusion use cases to implement an abstract use case	Abstract Extended UC to Concrete, Inclusion to Generalization
Functional Decomposition: using the include relationship [broken modularization]	Drop Functional Decomposition having Inclusion
Functional Decomposition: using the extend relationship [missing modularization]	Split Extension UC
Multiple generalizations of a use case	Generalization to Include
Use cases containing common and exceptional functionality	Drop Inclusion, Drop Extension
Multiple actors associated with one use case	Generalize Actors, Split UCs
An association between two actors	Drop Actor-Actor Association
An association between use cases	Drop UC-UC Association
An unassociated use case	Drop Unassociated UC
Two actors with the same name	Rename Actor
An actor associated with an unimplemented abstract use case	Abstract to Concrete, Add Concrete UC

It can be observed from the discussion provided in this section that the related literature has not yet studied the code and model smells from a security point of view. The literature studies mostly focus on proposing more bad smells or studying the impact of existing bad smells on the quality of source code and UML models. Some researchers are conducting studies to compare different bad smells in order to rank bad smells, in terms of the degree to which they negatively impact software quality attributes. Another concern to be noticed is the lack of work on proposing bad smells in use case diagram and sequence diagram. The class diagram has been the center of gravity for most the researchers to study bad smells and their impacts on software quality. These gaps in the literature stimulate our motivation to study bad smells from a security perspective and include other models such as use case diagram and sequence diagram as well.

3.3. Security Aspects in Software Development

This section provides a detailed description of methodologies aiming to solve the problem of security in terms of information systems modeling and development. This section also summarizes the security metrics, related to code and models, proposed in the literature.

Vivas et.al used the guidance of business model to take system development decisions [64]. The role of basic security components motivated them to study the business perspective of technology development. The purpose of integrating UML with security was to provide a standard modeling language incorporating security characteristics. Use cases were used to elicit security requirements and then they are injected to the functional specification. The process is iterated multiple times to ensure the presence of maximum security requirements. The final specification can then be used in proceeding stages of development. Jurgens aimed to specify security requirements to ensure confidentiality and integrity in UML [10]. The basic purpose was to propose an extension to help develop secure systems. Types of attacks were modeled by analyzing the behavior of attackers. Their study covered UML diagrams including state chart diagram, sequence diagram, deployment diagram.

Siponen and Baskerville provided a new paradigm to software developers to securely develop information systems [65]. To discover security design patterns, the author used analytical process having multiple phases. The first phase focused on finding common objects in software development and security development. The second stage included identification of security constraints, abuse cases and scenarios, and policies. In the final phase, the author gathered expert views on proposed patterns. After the consultation phase, the six elements were added to the meta-model.

Artelsmair et.al worked on the integration of security concerns with software modeling [66]. Security policies define rules and practices to manage and protect sensitive data. First, they showed how the defined rules and practices can be integrated into the modeling process. Secondly, they identified security requirement and corresponding security mechanism. They applied use cases to cater security requirements for modeling purpose. Fernandez emphasized on the application of security principles at every development stage [67]. At requirements stage, use cases can be used to express security requirements. Design stage can incorporate security concerns pointed in requirements stage. The defined security constraints can later be implemented at implementation phase. The inclusion of audit at the end of each stage further strengthens the security aspects.

The software metrics discussed in the background section of this thesis cover multiple quality attributes, applicable at design and implementation levels. Alshammari et.al presented security metrics that are restricted only to class diagram [11, 12]. They presented security metrics from five different perspectives: composition, coupling, extensibility, inheritance and design size. Each category further provides a set of metrics that address the security concern in software design. The presented metrics aimed at security concerns for class diagram only. The focus of class level security metrics is mainly from an accessibility point of view, for example, data accessibility, operation accessibility etc. The reported security metrics by Alshammari et.al [11, 12] are summarized in Table 4, with brief elaboration.

Table 4. Class level security metrics

Security Metrics	Description
Composite-Part Critical Classes	The ratio of the number of critical composed-part classes to the total number of critical classes in a design.
Critical Classes Coupling	The ratio of the number of all classes' links with classified attributes to the total number of possible links with classified attributes in a given design.
Critical Classes Extensibility	The ratio of the number of the non-finalized critical classes in a design to the total number of critical classes in that design.
Classified Methods Extensibility	The ratio of the number of the non-finalized classified methods in a design to the total number of classified methods in that design.
Critical Superclasses Proportion	The ratio of the number of critical superclasses to the total number of critical classes in an inheritance hierarchy.
Critical Superclasses Inheritance	The ratio of the sum of classes which may inherit from each critical superclass to the number of possible inheritances from all critical classes in a class hierarchy.
Classified Methods Inheritance	The ratio of the number of classified methods which can be inherited in a hierarchy to the total number of classified methods in that hierarchy.
Classified Attributes Inheritance	The ratio of the number of classified attributes which can be inherited in a hierarchy to the total number of classified attributes in that hierarchy.
Critical Design Proportion	The ratio of a number of critical classes to the total number of classes in a design.
Classified Instance Data Accessibility	The ratio of the number of classified instance public attributes to the number of classified attributes in a class.
Classified Class Data Accessibility	The ratio of the number of classified class public attributes to the number of classified attributes in a class.
Classified Operation Accessibility	The ratio of the number of classified public methods to the number of classified methods in a class.
Classified Methods Weight	The ratio of the number of classified methods to the total number of methods in a given class.
Classified Mutator Attribute Interactions	The ratio of the number of mutators which may interact with classified attributes to the number of mutators which could interact with classified attributes.
Classified Accessor Attribute Interactions	The ratio of the number of accessors which may interact with classified attributes to the possible maximum number of accessors which could have access to classified attributes.
Classified Attributes Interaction Weight	The ratio of the number of all methods which may interact with classified attributes to the total number of all methods which could have access to all attributes.

Another study with the purpose of proposing security metrics is conducted by Chowdhury et.al [68]. Their study aimed to provide security metrics for source code, including, stall ratio, coupling corruption propagation and critical element ratio. Stall ratio is computed as a ratio of a number of non-progressive statements in a loop to total lines in the loop [68]. This explains the hurdles which some statements may create to accomplish a program's goal. The second metric is coupling corruption propagation, which is calculated as the number of child methods instantiated with the parameters based on the parameters of original instantiation [68]. In object-oriented systems, some objects need to be instantiated at a specific time, and if not, they may destabilize the whole running process. This aspect of object invocation is named as critical element ratio and is computed as a ratio of critical data elements in an object to total elements in the object [68].

It can be observed that a handful collection of security metrics is obtained from the literature. The applicability of security metrics, listed in Table 4, is limited to class diagram only. Since our research aims to cover use case diagram and sequence diagram as well, the security aspects are to be covered by security attributes and requirements.

3.4. Bad Smells Detection Techniques and Tools

This section provides a detailed discussion on the detection techniques and tool support provided for different bad smells. The illustration of automatic, semi-automatic and manual detection approaches is provided in forthcoming sections. To completely eradicate the validity threat posed by manual detection, our work is focused on automated detection of bad smells. The fatigue and inefficient consumption of time also motivated us to execute our methodology automatically.

Bad smells can be detected by analyzing source code statically or dynamically. Static analysis is feasible because it does not require execution of code. Static analysis can be conducted by textual analysis or graphical analysis. A textual analysis of code vastly depends on granularity, for example, token level, character level, line level and method level. There are many detection mechanisms that can be used to detect bad smells and in this regard, one classification of detection technique is provided by Bhalla [69]. The techniques are classified into five classes: code auditing, software metrics, abstract syntax tree, software visualization and anti-patterns. Code auditing statically analyzes the code and check for anomalies. The classification which we consider in our research is presented by Misbhauddin and Alshayeb [6]. They classified three detection strategies namely: design patterns, software metrics and pre-defined rules [6]. The elaboration of detection techniques in terms of this classification is presented in the following sections.

3.4.1. Software Metrics Based

Software metrics provide statistical information about software artifacts by capturing the key attributes of them. The most famous object-oriented metrics reported in the literature are proposed by Chidamber and Kemerer [31]. The software metrics usually cannot be directly applied to UML models, hence models are first transformed into XML and then XML representation is parsed to measure software metrics. If the measured metrics values are not in acceptable range, it is considered as a bad smell. Hence, the major concern in metrics based techniques is the acceptable threshold values. Table 5 lists the studies which incorporated software metrics. Each study shows the consideration of metrics from different categories i.e. coupling, cohesion, complexity and inheritance. It can be observed that all classifications are studied evenly in the literature.

Table 5. Software metrics incorporated in metrics based bad smell detection techniques

Author(s)	**Metrics Classification**			
	Coupling	**Cohesion**	**Complexity**	**Inheritance**
Arendt and Taentzer [70]	✓	✓	✗	✓
Fourati et.al [71]	✓	✓	✓	✓
Moha et.al [72]	✓	✓	✓	✓
Ghannem et.al [7]	✓	✓	✓	✓
Van Gorp et.al [73]	✓	✓	✗	✓
Ruhroth et.al [74]	✗	✓	✗	✓
Saeki and kaiya [75]	✗	✗	✓	✗
Mohamed et.al [76]	✗	✗	✓	✗
Jensen and Cheng [77]	✓	✗	✓	✓
Enckevort [32]	✓	✓	✓	✓
Kempen et.al [78]	✓	✗	✗	✗

Arendt and Taentzer used model metrics and model smells to propose a detection process [63]. They presented an integration study of two tools, EMF Smell and EMF Refactor. The integration allows automatic detection and removal of model smells by applying suggested refactoring in class diagram and use case diagram. The goal of EMF Smell is to identify model smell against meta-model, presenting them in an understandable view. EMF Refactor consists of three main components: code generation module, refactoring application module and EMF model refactoring suite.

Fourati et.al proposed an approach to identify anti-patterns at the structural and the behavioral levels through the use of quality metrics [71]. The structural and behavioral level models considered in their study were class diagram and sequence diagram respectively. The basic purpose of incorporating sequence diagram was to compensate the loss of information, when moving from the source code to the design. The approach carries few steps. First, the relationship between bad smells and metrics is unveiled. Detection is done by transforming class diagram using XMI and then the software metrics are used to identify whether diagrams carry bad smells or not. Kempen et.al also worked on the preservation of model behavior as a result of refactoring [78]. They used the class diagram for model transformation purpose and state chart diagram to preserve behavior. They examined metrics specific to a design and suggested refactoring accordingly. Van Gorp et.al pointed out the problem of many UML tools about maintaining consistency, while refactoring model or source code [45]. They proposed an extended UML meta-model to rectify the consistency problem. They accomplished it by stating pre and post conditions to help verify refactoring, verify behavioral preservation and automatic triggering of bad smells refactoring.

Moha et.al provided a method that assists in specification and detection of bad smells at the class level and developed a detection technique that automatically executes their method [72]. A classification in terms of metrics relation with bad smells is provided as follows: Blob: controller class, controller method, low cohesion, large class, data class; Swiss army knife: multiple interface; Functional decomposition: private field, class one method, procedural names, no inheritance, no polymorphism; Spaghetti code: use global variable, no parameter, long method, no inheritance, procedural names, no polymorphism.

Saeki and kaiya proposed an integrated technique of software metrics and model driven development [75]. They specified meta-model for a class diagram to specify constraints. To be more precise, their focus was on following aspects: 1) Utilization of meta-modeling to propose model metrics; 2) Proposition of semantic model metrics, and 3) Specification method for the model transformation metrics. Mohamed et.al also presented an extended meta-model of UML to assist model driven refactoring [76]. Their approach allows automated detection of bad smells in class and sequence diagrams by the use of model metrics and design smells. They performed domain analysis to propose UML extended meta-model to achieve their objective. Once the design smells are identified, the proposed meta-model is applied to cater possible refactoring. Refactoring tags are assigned to the source model, indicating the need of restructuring. Before appropriate refactoring is applied, the user validates the refactoring tags.

Enckevort established a prototype that detects different aspects of model features to identify model improvement opportunities [32]. The prototype is based on model metrics, syntactic and semantic rules. Their established prototype is applied to class level design issues. Ruhroth et.al made use of quality cycle with repeated steps of detecting and refactoring bad smells at the class level [74]. They applied quality cycle in the domain of software models. Ghannem et.al proposed an approach to automatically detect class level refactoring opportunities by the application of the genetic algorithm [7]. Their approach exploits class diagram defects and search based technique to create rules that find defects in models.

3.4.2. Design Patterns Based

Anti-patterns, which is opposite of design patterns, is another technique used for bad smells detection purpose. Design patterns provide good solutions of defects in software development, on the other hand, anti-patterns indicate bad solutions to problems. They allow developers to identify common design and implementation problems and provide an appropriate solution. Improving design quality attribute in models by incorporating pattern into a design is called pattern based model refactoring [79]. The refactoring procedure based on patterns involves three stages: the setting of the source, setting of target model and applying transformation [79]. The part of software artifact which needs refactoring is first selected, then based on a design pattern, a target model is set. The selected portion of artifact is transformed in accordance with the defined target model. Bouhours et.al proposed an inspection procedure to detect bad smells in class diagram through the use of design patterns [80]. They introduced the term ‘spoiled pattern’, which provide inadequate solutions for any problem. To find parts of a model, substitutable with design patterns, their method parses the model to identify the parts which have the possibility of having bad design practices.

Kim defined design pattern consisting of three components: problem models, solution models and transformation models [81]. The transformation model describes how problem specification can be transformed to a solution specification. A problem specification is assessed against a specific design pattern for its applicability on that problem. If the pattern specification matches with problem specification, the corresponding transformation model is applied. They provided refactoring specifications for Abstract Factory pattern, Adapter pattern and Observer pattern.

Moghadam and Cinneide presented a refactoring approach considering program’s design and source code [34]. The developer creates the desired design for a particular program, then the code is modified to complement the desired design. Their approach improves source code having better design attributes without affecting the behavior. Jensen and Cheng applied design patterns to cope with class level bad smells [77]. The proposed approach applied genetic algorithm and software metrics to identify the suitable refactoring that can be applied to design smells. Their automated approach was able to generate refactoring based on software metrics. Song et.al proposed a new notation to specify pattern solution called Role Models [79]. They emphasized on the use of Role Models on pattern based refactoring for the class diagram. The abstract factory was considered to describe the working of their technique.

3.4.3. Rule Based

Rule-based techniques ensure the use of a specific template or standard rules to develop software artifacts. If a software artifact is not created using a predefined standard, it is suspected to have bad smells. For example, El-Attar and Miller provided a template to specify use cases and relevant descriptions [14]. They reported that their proposed template enhances consistency in use case diagram and their descriptions. They presented a semi-automated technique based on anti-patterns to provide remedies for common quality problems in use case diagram. The technique provides a framework to define anti-patterns. They claimed that application of their proposed technique would transform use case model into a more accurate representation of functional requirements. They also provided a repository of anti-patterns, containing 26 domain independent anti-patterns. Using the same taxonomy of anti-patterns, Khan and El-Attar proposed a technique to refactor the specified anti-patterns [16]. They used model transformation approach using OCL to detect and refactor use case diagrams. Rui and bulter described the application of refactoring on use cases [82]. They formulated a meta-model for use case diagram. The extended use case meta-model includes Inclusion, extension, generalization, precedence, similarity and equivalence.

Dobrzanski and Kuzniarz presented an approach to systematically specify bad smells and associated refactoring in class and sequence diagrams [83]. A template is used which includes information: name of refactoring, origin, trigger element, goal, reasons, bad smell, pre and post conditions. They considered UML models built in TAU CASE tool. Llano and Pooley studied the specification and correction of anti-patterns related to the class diagram [84]. They defined the UML-based specification of anti-patterns and corrected them via application of design transformations. Sunye et.al worked on the application of certain rules to ensure the preservation of design behavior after the application of refactoring techniques [85]. They presented a few refactoring opportunities with the objective of understanding how they can be used to preserve behavior in class diagrams. Boger et.al presented a browser for refactoring which is integrated with UML modeling [47]. The applicability of their approach is validated on class diagrams.

Few observations can be made from the explanations of presented detection techniques. Table 6 summarizes all the surveyed detection strategies and provides a comprehensive overview of model sources incorporated in each study, alongside detected bad smells and tool support. The detection of bad smells in class and sequence diagrams is accomplished via design patterns, software metrics and pre-defined rules. Model smells in use cases are detected using metrics and pre-defined rules. The class diagram is the most investigated UML model in the context of model smell detection. The detection of model smells in class diagram is mostly supported by tools. The major subset of model smells is studied for class diagrams only. Sequence diagrams are studied in conjunction with class diagram. The reason is the similar type of model smells for both diagrams and the way they are detected. It can be observed that literature studies are leaned towards detection and refactoring of bad smells in class diagrams. Few studies address the issues related to refactoring in use case diagram and sequence diagram. The researchers have also not yet addressed detection and refactoring of bad smells from a security perspective.

Table 6. Summarization of surveyed detection techniques

Author(s)	**Technique**	**Model**	**Bad Smells**	**Tool**
Arendt and Taentzer [70]	Metrics based	Class, use cases	Missing abstraction, long parameter list, unused use case, unassociated classes, two subclasses with same field, data clumps.	Yes
Fourati et.al [71]	Metrics based	Class, sequence	Blob, lava flow, functional decomposition, poltergeists, swiss army knife.	No
Dobrzanski and Kuzniarz [83]	Rule based	Class, sequence	Middle man, same methods in subclasses, unused operation, inappropriate method signature.	Yes
Moghadam and Cinneide [34]	Design pattern	Class	Unnecessary hierarchy, missing abstraction, excessive delegations, field/method is used by some subclasses, same fields/methods names, un-encapsulated field/method.	Yes
Bouhours et.al [80]	Design pattern	Class	Not specified.	Yes
Moha et.al [72]	Metrics based	Class	Blob, functional decomposition, spaghetti code, swiss army knife.	Yes
Ghannem et.al [7]	Metrics based	Class	Blob, functional decomposition, data class.	No
Van Gorp et.al [73]	Metrics based	Class	Two subclasses with the same method, un-fragmented code.	Yes
El-Attar and Miller [14]	Rule based	Use cases	See Table 3.	Yes
Khan and El-Attar [16]	Rule based	Use cases	See Table 3.	No
Ruhroth et.al [74]	Metrics based	Class	Hidden concurrency, unnecessary behavioral complexity, low cohesion, strong coupling, refused bequest.	Yes
Saeki and kaiya [75]	Metrics based	Class	Not specified.	No
Mohamed et.al [76]	metrics based	Class, sequence	Blob.	Yes
Jensen and Cheng [77]	Metrics based	Class	Abstract access, delegation, encapsulated construction, partial abstraction.	Yes
Boger et.al [47]	Rule based	Class	Improper class name, two subclasses with the same method.	Yes
Enckevort [32]	Metrics based	Class	God class, cyclic dependency, poor use of abstraction, encapsulate field, long parameter list, data class.	Yes
Sunye et.al [85]	Rule based	Class	Inappropriate method signature, unused attribute/method/class, feature envy, missing modularization, unutilized abstraction.	Yes
Rui and bulter [82]	Rule based	Use cases	Absent use case, unused use case, improper use case name, moving an element of use case.	No
Kim [81]	Design pattern	Class, sequence	Not specified.	Yes
Kempen et.al [78]	Metrics based	Class	God class.	Yes
Llano and Pooley [84]	Rule based	Class	God class, poltergeist.	No
Song et.al [79]	Design pattern	Class	Inheritance smells.	No
Ouni et.al [8]	Rule based	Class	Blob, functional decomposition, spaghetti code.	No

3.4.4. Detection Tools

The effective and commonly used detection tools are collected from the literature. The summary of identified tools alongside additional information related to bad smells, type, code linkage and language support is presented in Table 7. Different bad smells can be collected by applying different tools. iPlasma and inFusion handle a bigger subset of bad smells in comparison with other detection tools. It can also be noticed that the tools which support code linkage are offering a fewer subset of bad smells. Some tools require Eclipse to execute, whereas majority can work standalone. The reason is the wide use of Eclipse as a development platform. Java, being the most popular object oriented modern language, is supported by many tools, while few tools support other languages such as C and C++. While detecting a bad smell from code, few tools even point to the location from where bad smell is originated, making refactoring easier.

Table 7. Tools for automated detection of bad smells

Tool	Smell detection	Type	Code linkage	Language Support
Checkstyle [86]	Duplicated code, large class, long method, long parameter list	Eclipse, standalone	Yes	Java
Décor [87]	Data class, god/large class, long method, long parameter list, message chain, refused bequest, speculative generality, tradition breaker.	Standalone	No	Java
iPlasma [88]	Brain class, brain method, data class, duplicated code, extensive coupling, feature envy, intensive coupling, refused bequest, shotgun surgery, tradition breaker	Standalone	No	C++, Java
inFusion [89]	Brain class, brain method, data class, data clumps, duplicated code, extensive coupling, feature envy, intensive coupling, refused bequest, shotgun surgery, tradition breaker	Standalone	No	C, C++, Java
JDeodorant [90]	Feature envy, god/large class, long method, switch statements	Eclipse	Yes	Java
PMD [91]	Dead code, duplicated code, large class, long method, long parameter list	Eclipse, standalone	Yes	Java
Stench blossom [92]	Data clumps, feature envy, large class, long method, message chains, switch statement, typecast	Eclipse	Yes	Java

CHAPTER 4

RESEARCH METHODOLOGY

This chapter highlights the major aspects of our research methodology. The main research goal that our methodology includes how efficiently our detection and correction approaches are able to respectively detect and correct security bad smells. Another important research goal posed for our research methodology is the evaluation of security improvement as a result of refactoring to security bad smells. Although surveying for security bad smells and quality metrics are performed, the focus of this section is on filtering the security bad smells and related refactoring strategies; detection and correction of security bad smells; and evaluation of security improvement in studied UML models as result of refactoring.

4.1. Research Methodology Overview

Although all the main activities of our implemented research methodology are illustrated in detail in forthcoming section, the purpose of this section is to provide the reader with a basic overview of our methodology. To further ease the understandability, the pictorial view of our research methodology is depicted in Figure 4.

The first activity in research methodology is the filtration of security bad smells. A large taxonomy of bad smells exists in literature, which can undergo some filtration process to strain only security bad smells. Once the security bad smells are successfully filtered, they and related quality metrics are input to the GA for the purpose of detection of the smells. As a result, the GA yields detection rules. The generated detection rules are applied on

UML models to detect existing bad smells in them. The rules use combinations of conditions to detect bad smells. This accomplishes the detection objective. The next objective focus on correction of detected security bad smells. The investigated UML models are transformed using XMI. Considering the detected security bad smells, the refactoring strategies are applied to XML representations of UML models, which results in the generation of refactored XML representations. The refactored XML representations are exported back to corresponding refactored UML models. The refactored UML models are processed using post refactoring conditions to ensure behavioral preservation. The quality metrics are computed, before and after refactoring, using XML representations of UML models. The comparison of quality metrics pre and post refactoring assists in assessing the quality improvement of UML models from a security perspective. The basic flow of activities is shown below in Figure 4.

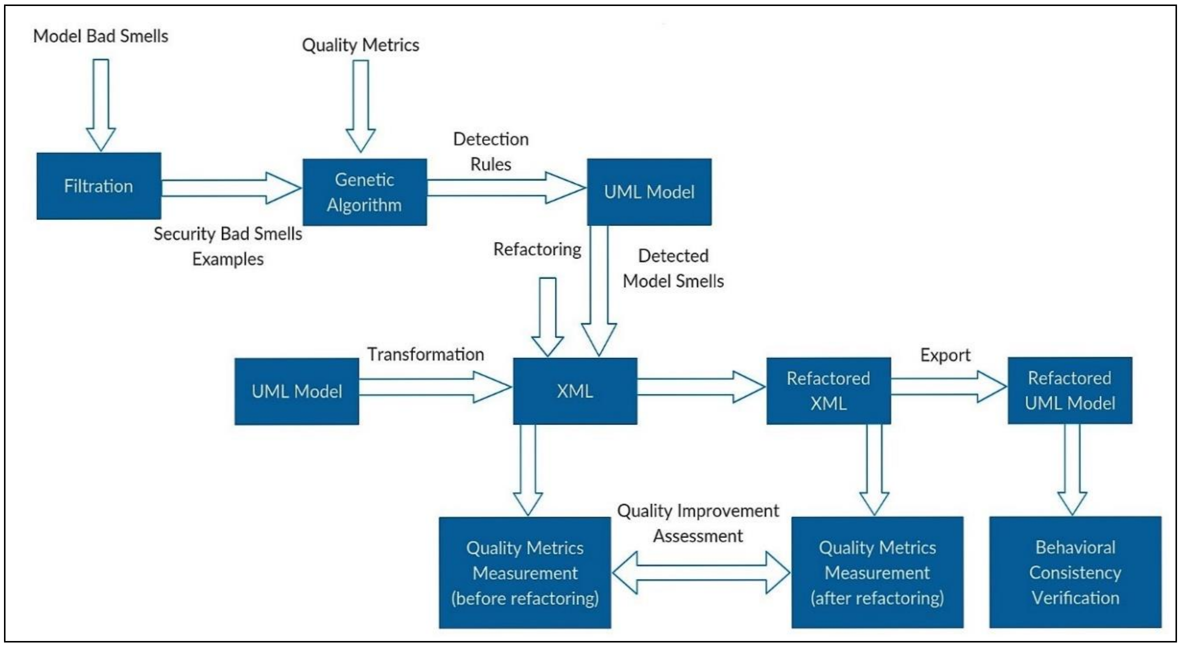
Model Bad Smells
Quality Metrics
Filtration
Genetic Algorithm
Detection Rules
UML Model
Security Bad Smells Examples
Refactoring
Detected Model Smells
UML Model
Transformation
XML
Refactored XML
Export
Refactored UML Model
Quality Improvement Assessment
Quality Metrics Measurement (before refactoring)
Quality Metrics Measurement (after refactoring)
Behavioral Consistency Verification

Figure 4. Research Methodology Overview

4.2. Filtration of Security Bad Smells

The tagging of existing bad smells as security bad smells is a non-trivial task. It needs to be assured that bad smells tagged as security bad smells violates one or more security attributes. The filtering process is eased by the study of Suryanarayana et.al, in which they classified many design smells [29]. Besides design smells classification, they also identified the quality attributes each design smell tarnish. They also reported few violations of security attributes by some design smells. This allows us to filter the model bad smells which violate security attributes and safely tag them as security bad smells.

The existing catalog of model bad smells is used to analyze which model bad smell is affecting the security attributes. If a bad smell from existing catalog violates any security attribute, it is tagged as a security bad smell. For example, missing modularization is a model bad smell reported by Suryanarayana et.al, and they identified understandability, changeability, extensibility, reusability, testability and reliability as affected quality attributes because of the presence of this bad smell [29]. According to the definitions provided for information security (presented in section 2.2), reliability is one of the security attributes. Hence, missing modularization can be filtered as a security bad smell. Similarly, other security bad smells are filtered. This was an example of a security bad smell. The forthcoming example identifies a bad smell as a non-security bad smell using the same procedure. According to Suryanarayana et.al, imperative abstraction is a design smell and impacts understandability, changeability, extensibility, reusability and testability [29]. Since none of these quality attributes are related to security as per the security definitions (presented in section 2.2), this bad smell is not filtered by our process. In a similar manner, other non-security bad smells are identified. The brief definition of each security bad smell, along with the security requirements it violates, and appropriate refactoring, are presented in Appendix C.

In the scope of our research, we are focusing on three security bad smells in each model. Table 8 lists the security bad smells considered in each model. The brief definition of each security bad smell, along with the security requirements it violates, and appropriate refactoring, are presented in Appendix C. All three models have two (missing and broken modularization) common security bad smells and one different security bad smell. Multiple instances of same bad smell are ensured to have diversity in our solution.

Table 8. Investigated security bad smells in each model

Use case diagram	Sequence diagram	Class diagram
Missing hierarchy	Missing modularization	Missing modularization
Missing modularization	Broken modularization	Broken modularization
Broken modularization	Unutilized abstraction	Deficient encapsulation

4.3. Detection Approach

The gaps identified in the literature are addressed by our proposed detection and correction approaches. The idea of detection approach is inspired by the technique presented by Ouni et.al [8], with changes lie in GA process, specifically for crossover and mutation operations. Another distinction lies in the consideration of security bad smells rather than normal class bad smells. The different set of studied bad smells enforces the use of a different set of quality metrics. In addition, our major contribution resides in the application of our approach on use case diagrams and sequence diagrams.

4.3.1. Approach Overview

The detection rules are generated using security bad smells examples through the application of a genetic algorithm. This step takes bad smells examples and quality metrics as inputs and generates a set of rules. The detection rules use a set of metrics and their values to detect a specific defect. The quality metrics values are collected automatically through SDMetrics tool [93], except for sequence diagrams. The set of metrics converging to a bad smell is used as a rule for the detection of that specific bad smell. The best-fitted solution, obtained from genetic algorithm application, carries the set of rules which detects a maximum number of bad smells. For instance, the following rule identifies whether a given class is a blob or not.

Rule: if (numberOfAttributes > 10 AND numberOfMethods > 20 AND KLOC > 5000) Then Blob.

In this example, these three metrics measure a class to be a blob or not. If the metrics values of a given class exceed the values specified in the above rule, the blob is detected.

4.3.2. GA adaptation to Detection Approach

This section demonstrates the application of genetic programming in the context of bad smells detection. Genetic programming is a heuristic search based approach based on the Darwinian theory of evolution [94]. It explores the search space to find a best-fitted solution for a specific problem. The definition of the following elements is necessary to apply genetic programming to current problem:

- Individual formulation.
- Population creation from individuals.

- Fitness function computation to evaluate the fitness of an individual in solving the problem.
- Selection of individuals for the creation of new population.
- New individual creation through crossover and mutation for the purpose of exploring search space.
- New population generation.

The abstract view of our applied genetic algorithm is summarized below in Figure 5. The algorithm takes quality metrics and security bad smells examples as inputs and yields the best solution that corresponds to a set of detection rules that best detect the bad smells in models. Lines 1-2 forms the initial population of a genetic algorithm; comprising of individuals. An individual is represented by a set of rules with corresponding bad smells. The set of all individuals formulates a population. Lines 4 -13 represents the main genetic algorithm loop. It explores the search space and constructs new population. The quality of individuals is evaluated in each iteration. The expression in line 9 saves the individual carrying best fitness. The new population is generated by selecting the comparatively best-fitted individuals from existing population and then exposed to crossover and mutation operations. During crossover, the selected pair of parents produces two new individuals. The mutation operator ensures solution diversity in both parents and children. The algorithm terminates when an individual identifies maximum defects present in the given model. At the end, the algorithm returns the best solution containing rules that are capable of identifying maximum defects in a model.

```
Input:
Quality metrics
Security bad smells examples
Process:
1.  I = set of rules
2.  P = set of I
3.  M = model
4.  repeat
5.      for all I in P do
6.              detected bad smells = execute_rules(M);
7.              fitness (I) = numberOfDetectedBadSmells;
8.      end for
9.      best_solution = best_fitness(I);
10.     P = new_population(P);
11.     it = it + 1;
12. until it = max_bad_smells;
13. return best_solution;
Output:
best_solution
```

Figure 5. A high-level GA adaptation for detection

a) Individual and population representation

An individual is comprised of a set of rules having IF-THEN statements. The expressions in the rules are a combination of OR and AND logical operators. IF statement executes the conditions with quality metrics to detect a bad smell and if the IF statement returns true, the corresponding bad smell exists in the model. Each individual is composed of three rules with each rule is exploring for a specific bad smell. An instance of an individual representation containing rules for bad smells detection in the sequence diagram is shown in Figure 6. If the number of associations (NAss), the number of invocations (NInvoc), the number of received messages (NRec) and the number of coupled classes (CBO) of a class in sequence diagram equal or exceed the specified thresholds, then the specified security bad smell exists in the given sequence diagram. Only the description of the individual formulation is presented here, the definitions of the quality metrics are presented in chapter 5, while explaining variables.

R1: IF (NAss(c) >= 38 AND (NInvoc(c) >= 11 AND NRec(c) >= 19) AND CBO(c) >= 3) THEN missing modularization(c)
R2: IF (NAss(c) == 2 AND (NInvoc(c) == 0 OR NRec(c) == 1) AND CBO(c) == 1) THEN broken modularization(c)
R3: IF (NAss(c) == 0 AND NInvoc(c) == 0 AND NRec(c) == 0 AND CBO(c) == 0) THEN unutilized abstraction(c)

Figure 6. Individual representation

The number of individuals depends on the number of rules, which further depends on a number of bad smells. The initial population is formed by the union of all the individuals. The size of the initial population depends on a number of individuals. To reiterate, the greater the number of rules, the more the individuals can be formed. So the size of the population is indirectly contingent upon the quantity of rules.

b) Selection

For the purpose of crossover and mutation, the individuals need to be selected. The selection is based on the relative fitness of individuals. In each iteration, the fitness value is calculated for every individual and two-third of the relatively best-fitted individuals are selected. The rest one-third is discarded in each iteration. The discarded one-third of the population is regenerated from the selected two-third of the population through crossover and mutation.

c) Crossover

It is understood that an individual is composed of three rules with each rule focusing on a single bad smell. For a crossover, one of the three rules from an individual is randomly selected and swapped with the same bad smell rule in another individual. This way two new individuals are created. For example, if two individuals I1 and I2 are randomly selected for crossover, R1 in I1 will be swapped with R1 of I2. The swapping leads to the introduction of two new individuals I1` and I2`. I1` has R1 of I2, and R2 and R3 of I1, whereas, I2` has R2 and R3 of I2 and R1 of I1. This stipulates new children (I1` and I2`) for having information from both parents (I1 and I2).

d) Mutation

The purpose of mutation is to encompass diversity in solution. In our algorithm, the mutation is achieved by modifying the value of quality metrics. The algorithm randomly selects an individual, followed by a rule and then a metric, whose value will be changed. The modification in the metric value can be in the form of increase or decrease. The decision of either increasing or decreasing is also randomized. The metric value is either increased by one or decreased by one depending on the generated random value. For example, suppose the individual presented in Figure 6 is randomly selected, then R1 and then metric NInvoc. Random value generation also suggests to increase the metric value, then the mutation is achieved by adding one to the current value, making it 12.

e) Fitness evaluation

The quality of an individual is only indicated by how well the encapsulated rules have performed in detecting security bad smells. The definition of our applied fitness function is simple yet effective. Fitness function calculates the number of detected bad smells against the existing bad smells in a model. The fitness value of an individual is maximized if the rules belonging to that individual are able to detect all the defects present in a given model. If a rule is able to detect a bad smell, a value of one is added to its individual’s fitness and if the rule is unable to suspect a bad smell, zero is added to the fitness value. The more the rules, present in the individual, detect bad smells, the greater the fitness value is. The individuals having relatively greater fitness values are selected for crossover and mutation operations.

4.4. Correction Approach

The correction of security bad smells is achieved through model transformation. The considered UML models are first transformed using XMI, then quality metrics are extracted from them. The XML representations of UML model diagrams are corrected based on the related refactoring of the security bad smells identified in them. XML provides sufficient information about the transformed model. The information is presented in the form of tags, which makes information extraction convenient. The detected security bad smells in a UML model can be traced in the corresponding XML representation. The tags are then modified manually according to the refactoring techniques for the eradication of the detected smells. Once the refactoring is successfully applied, the corrected XML representations are exported back to corresponding UML models. This way, the collected UML models are no longer hosting security bad smells. The more description about the correction approach, using a use case diagram as an instance, can be found in section 5.2.4.

4.5. Behavioral Consistency

The consistency approach that we apply in our UML refactoring is post-condition based. It was mentioned while explaining refactoring process, that consistency can be checked via pre-conditions or post-conditions or both. In this research, we opt for post-conditions consistency approach. We formulate some conditions before refactoring and once the refactoring is performed, the conditions are validated. For instance, in a use case diagram, where refactoring strategy deletes a use case, it must be checked after refactoring whether the functionality still exists or not. The use case diagram should be skimmed to ensure the presence of that functionality. Usually deleting a use case refactoring is performed if the use case is an inclusion use case and is included by a single use case. The inclusion use case can be removed and the related functionality is assumed to be encapsulated in the included use case. For this instance, it must be validated that included use case contains the inclusion use case functionality. The illustrations about how corrections of security bad smells are validated in terms of behavioral consistency are presented in chapter 5 for each investigated UML model.

4.6. Security Improvement Validation

The assessment on security improvement in UML models as a result of refactoring is achieved through statistical analysis of quality metrics. The specified quality metrics are calculated pre and post refactoring for each UML model, allowing observing the change in metrics values. It is expected that the metrics values will change as a result of refactoring but the evidence of how significant the change is, can only be assessed through statistical analysis. For this purpose, the pair-wise t-test is chosen. The pair-wise t-test is beneficial in our case because it reflects the significant change in a pair of values. Since we have metrics values before and after refactoring, it is an appropriate statistical test to execute, which can conclusively suggest on significant improvement of security as a result of refactoring. The security improvement validation is performed in chapter 5 for each investigated UML model.

CHAPTER 5

MODEL REFACTORING TO SECURITY

The purpose of this chapter is to provide details on the application of our detection and correction approaches on considered UML models to achieve the goals of this research. This chapter also explains our experimental setup and presents the obtained results. The explanations related to experiments are presented accordingly to the guidelines provided by Jeditschka et.al [17]. Since our work covers multiple UML models (class diagram, sequence diagram and use case diagram), the illustrations of our approaches are presented separately for each model.

5.1. Experimental Goals

The main experimental goal is presented below in the form of GQM (Goal Question Metric) approach [95]. The goal is to:

"Analyze the model refactoring to security bad smells for the purpose of improving software quality with respect to security"

The achievement of the main goal can be broken down into multiple sub-goals. The sub-goals include successful detection and correction of security bad smells and to what extent refactoring can improve the software in terms of security. To reiterate, following are our research questions and our experiments aim to address them:

RQ1: To what extent can our proposed detection approach detect security bad smells in

UML models?

RQ2: To what extent can our proposed correction approach rectify security bad smells in UML models?

RQ3: To what extent can refactor to security bad smells improve security aspects of UML models?

The basic mechanisms to answer these research questions are as follows:

For RQ1, existing security bad smells examples along with quality metrics are used to evaluate the recall of our proposed detection approach.

For RQ2, the correction efficacy is computed in terms of how many security bad smells are eradicated by our correction approach.

For RQ3, we use the t-test statistical analysis of quality metrics.

5.2. Experimental Design for Use Case Diagrams

This section aims to provide details on our experiment with use case diagrams. The following subsections describe our experiment in terms of experimental materials, variables, proposed hypotheses, experimental tasks, results and hypotheses testing.

5.2.1. Experimental Materials

Four use case diagrams belonging to four different systems are used in experiments with use case diagrams. The investigated use case diagrams can be found online [96]. The selection of the use case diagrams is achieved through random sampling to avoid any biasness towards our results. The descriptive statistics about the four investigated use case diagrams are presented in Table 9. The statistics are presented in terms of number of use cases present in the system, the number of actors interacting with the system, number of include relationships and number of extends relationships. The numbers of use cases and actors are evenly balanced in selected use case diagrams.

Table 9. Statistics of investigated use case diagrams

System	Use cases	Actors	Includes	Extends
ATM system (Figure 7)	10	3	3	3
HR system (Figure 8)	8	4	3	4
Restaurant system (Figure 9)	13	4	1	10
Travel agency system (Figure 10)	9	5	1	6

As shown earlier in Table 8, the investigated security bad smells in the use case diagram are missing hierarchy, broken modularization and missing modularization. Multiple instances (total 27) of these three security bad smells can be seen in investigated use case diagrams. The multiple instances of same bad smell allow diversity in generated rules, which contribute to the solution’s effectiveness.

Figure 7 shows the use case diagram of an ATM system [96]. The system provides a variety of services mainly system maintenance, transaction, login etc. Three actors (Administrator, Customer and Bank) are interacting with the system. The system offers four types of transactions: deposit, balance check, withdraw and print receipt having generalization relationships. Transaction and system maintenance require actor’s login so include relationships are present. Both these use cases can also lead to the exceptional execution of ‘Bad Pin’ use case. The given ATM system contains three security bad smells, namely, missing hierarchy, broken modularization and missing modularization. The instances where these bad smells are present are listed as follows:

Missing hierarchy:

UB1: Actor 'Bank' is accessing 'System Reporting' extension use case.

Missing modularization:

UB2: 'Bad Pin' use case is extending two use cases: 'System Maintenance' and 'Transaction'.

Broken modularization:

UB3: 'System Shutdown' is included by just one use case 'System Maintenance'.

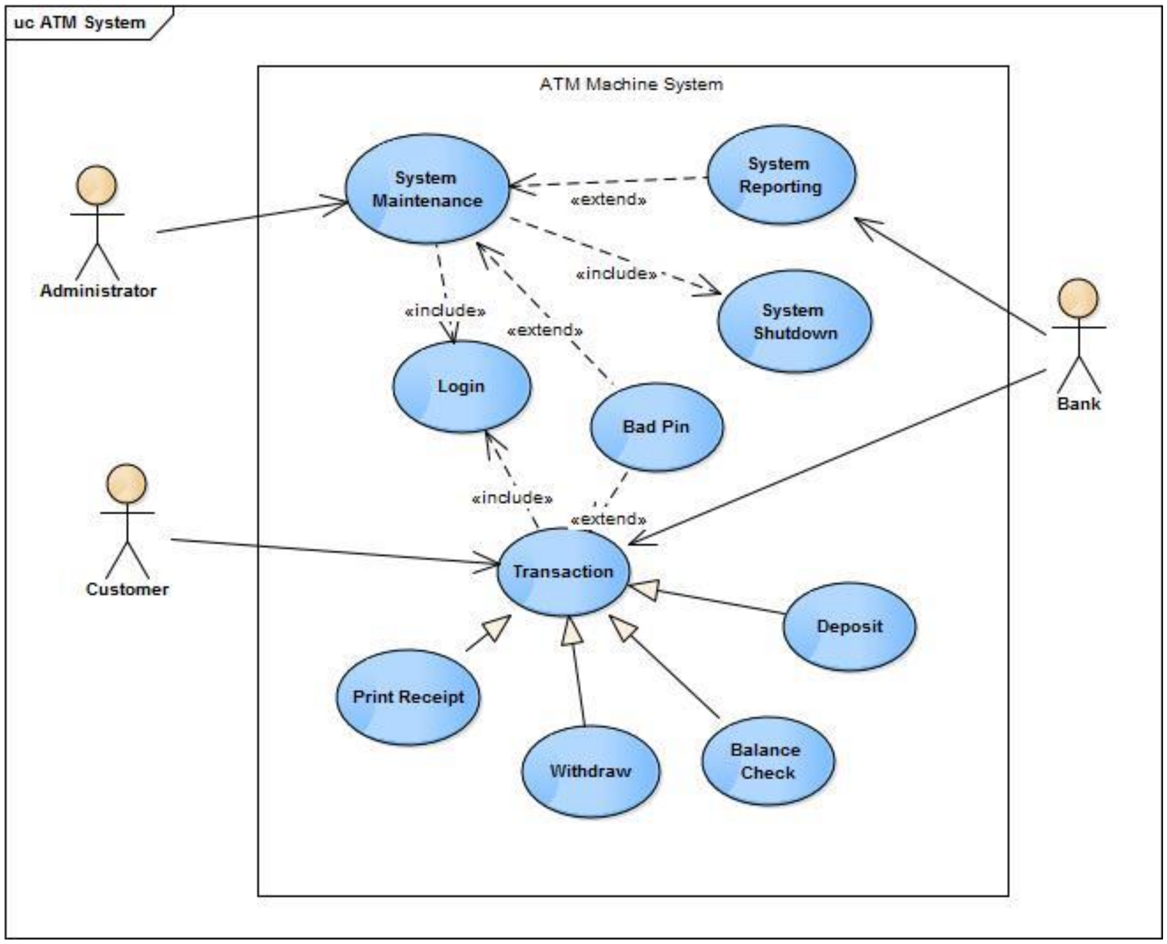

Figure 7. Use case diagram of ATM system

The use case diagram of human resource system is shown in Figure 8 [96]. It provides multiple services like update benefits, elect reimbursement from health care, elect stock purchase and issue purchase invoice. Besides getting benefits, Employee has the option of electing reimbursement and stock purchase. The stock entity is responsible for electing and purchasing of stock. The purchase invoice of stock is issued by an HR representative. The presented HR system contains three security bad smells: missing hierarchy, broken modularization and missing modularization. The instances where these bad smells occur are listed as follows:

Missing hierarchy:

UB4: Actor 'Health Care Dept.' is accessing 'Elect Reimbursement from Health Care' extension use case.

UB5: Actor 'Stock Entity' is accessing 'Elect Stock Purchase' extension use case.

Missing modularization:

UB6: 'Elect Stock Purchase' use case is extending three use cases: 'Update Benefits', 'Provides Stock' and 'Issue Purchase Invoice'.

Broken modularization:

UB7: 'Update Dental Plan' is included by just one use case 'Update Benefits'.

UB8: 'Update Insurance Plan' is included by just one use case 'Update Benefits'.

UB9: 'Update Medical Plan' is included by just one use case 'Update Benefits'.

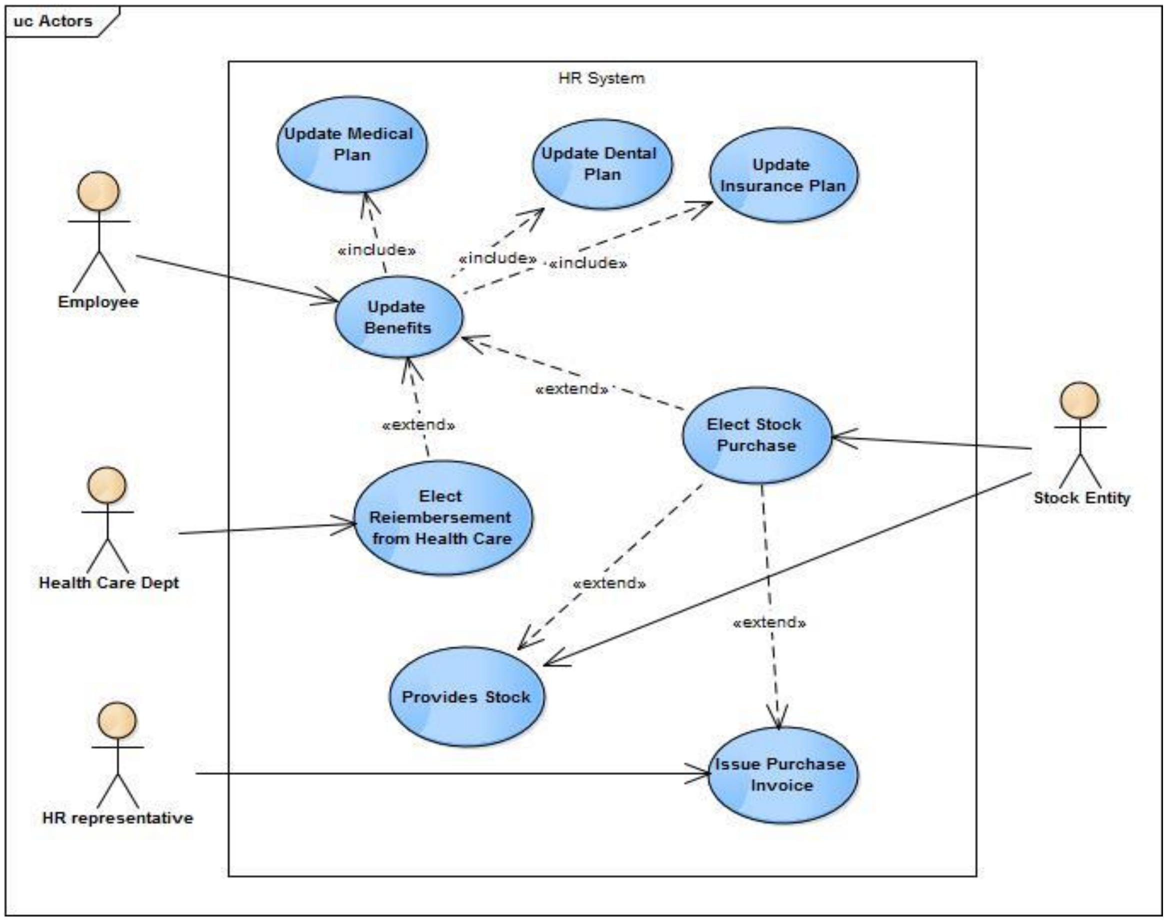

Figure 8. Use case diagram of HR system

Figure 9 shows the use case diagram of a restaurant system [96]. The main services offered by the system includes order food, cook food, serve food, eat food and pay for food. The presented restaurant system contains three security bad smells: missing hierarchy, broken modularization and missing modularization. The instances where these bad smells occur are listed as follows:

Missing hierarchy:

UB10: Actor ‘Waiter’ is accessing ‘Order Wine’ extension use case.

UB11: Actor ‘Customer’ is accessing ‘Order Wine’ extension use case.

UB12: Actor ‘Chef’ is accessing ‘Order Wine’ extension use case.

UB13: Actor ‘Waiter’ is accessing ‘Serve Wine’ extension use case.

UB14: Actor ‘Customer’ is accessing ‘Drink Wine’ extension use case.

UB15: Actor ‘Customer’ is accessing ‘Pay for Wine’ extension use case.

UB16: Actor ‘Waiter’ is accessing ‘Pay for Wine’ extension use case.

Missing modularization:

UB17: ‘Chinese’ use case is extending two use cases: ‘Order Food’ and ‘Pay for Food’.

UB18: ‘Italian’ use case is extending two use cases: ‘Order Food’ and ‘Pay for Food’.

UB19: ‘Indian’ use case is extending two use cases: ‘Order Food’ and ‘Pay for Food’.

Broken modularization:

UB20: ‘Pay tip’ is included by just one use case ‘Pay for Food’.

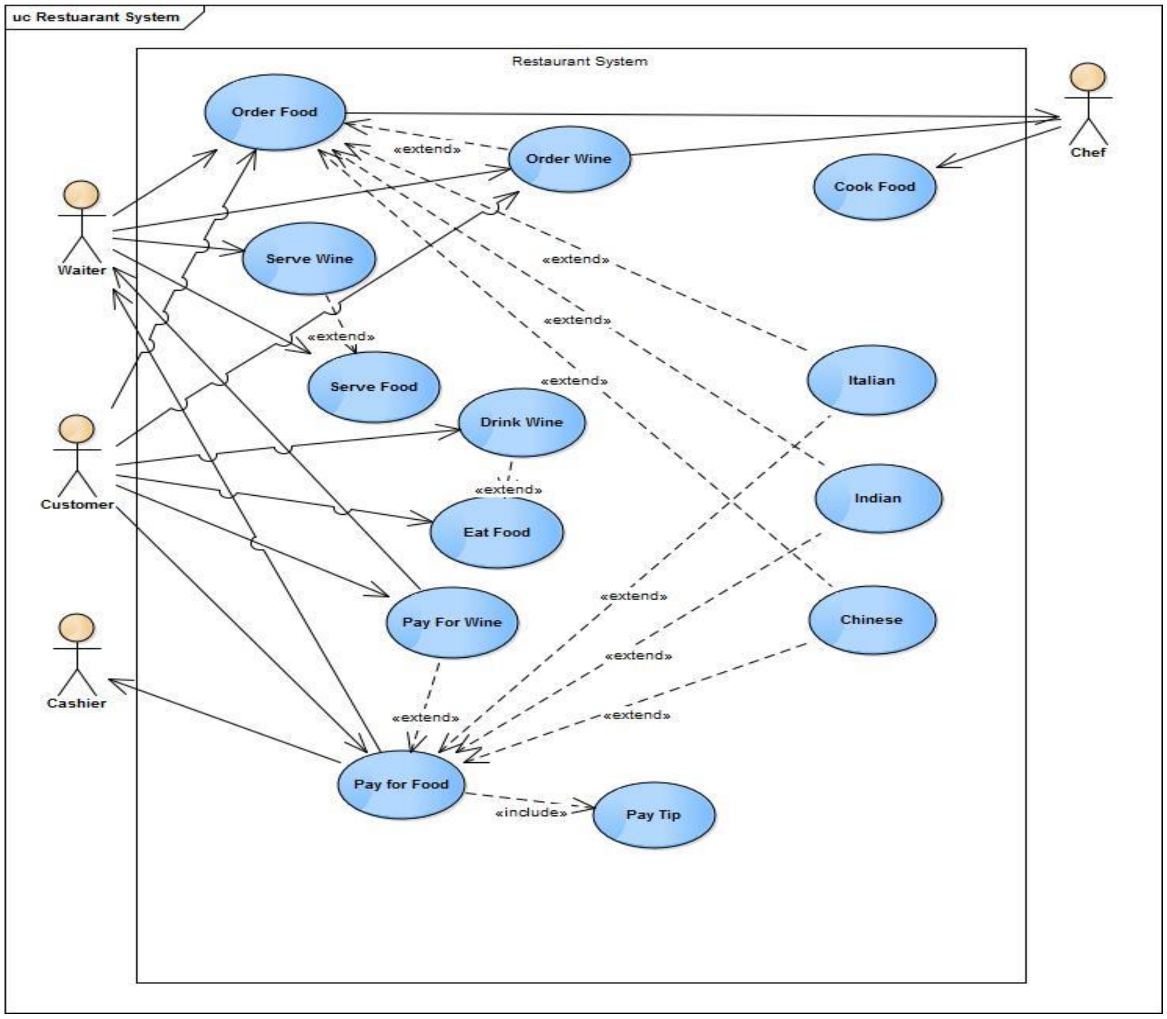

Figure 9. Use case diagram of restaurant system

The use case diagram of a travel agency system is presented in Figure 10 [96]. The system is responsible for booking and issuance of tickets and tours. The presented travel agency system carries three security bad smells: missing hierarchy, broken modularization and missing modularization. The instances where these bad smells occur are listed as follows:

Missing hierarchy:

UB21: Actor ‘Travel Agent’ is accessing ‘Book Airline Tickets’ extension use case.

UB22: Actor ‘Airline Company’ is accessing ‘Book Airline Tickets’ extension use case.

UB23: Actor ‘Client’ is accessing ‘Book Airline Tickets’ extension use case.

UB24: Actor ‘Airline Company’ is accessing ‘Pay for Airline Tickets’ extension use case.

Missing modularization:

UB25: ‘Book Airline Tickets’ use case is extending two use cases: ‘Book Tour’ and ‘Arrange Tour’.

UB26: ‘Pay Commission’ use case is extending three use cases: ‘Pay Travel Agent’, ‘Pay for Airline Tickets’ and ‘Pay for Tour’.

Broken modularization:

UB27: ‘Pay for Airline Tickets’ is included by just one use case ‘Pay Travel Agent’.

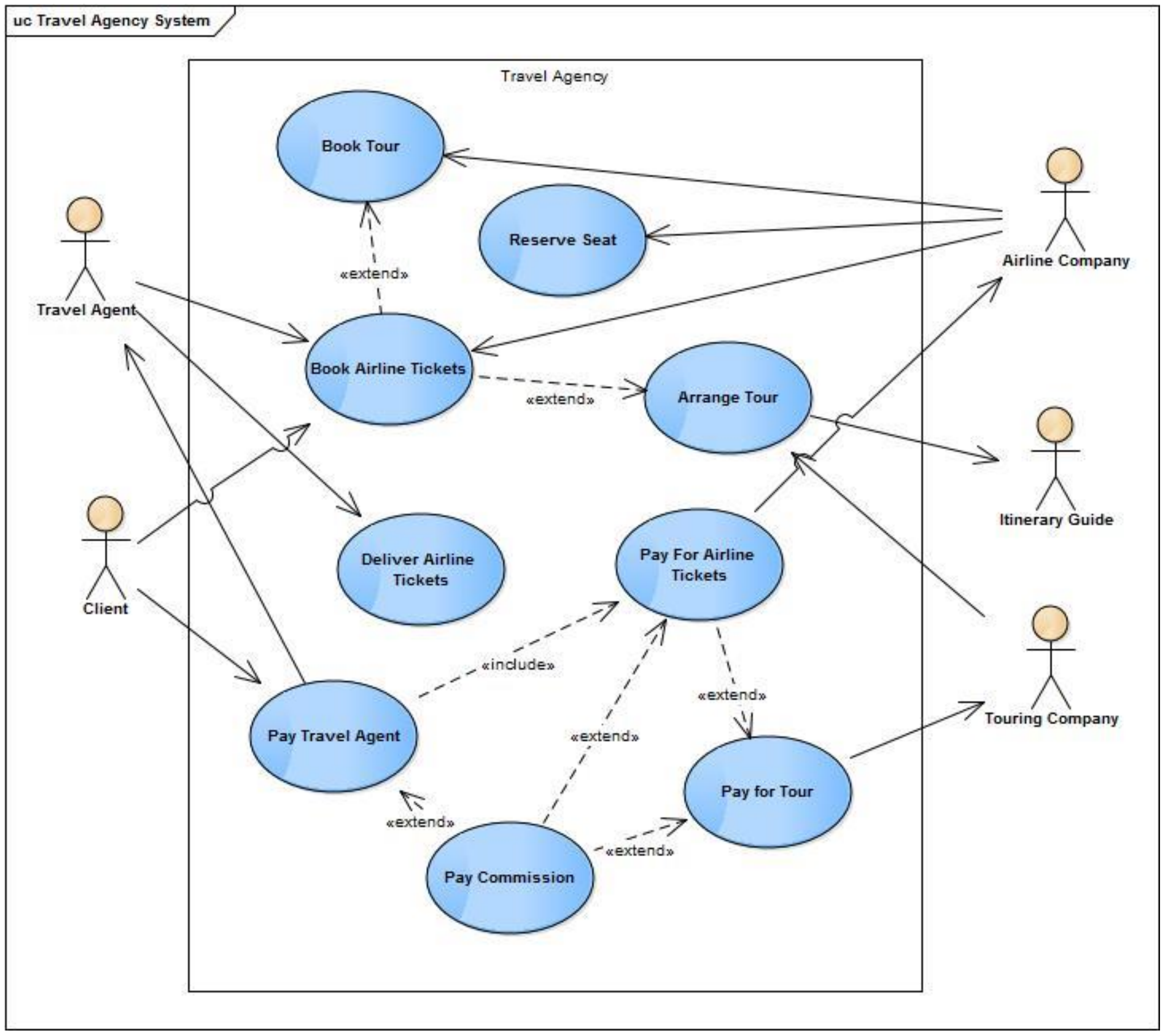

Figure 10. Use case diagram of travel agency system

The collection of quality metrics is accomplished through SDMetrics [93]. The construction of use case diagrams is performed using Enterprise Architect [97]. This tool is also useful in exportation and importation of use case diagrams to and from XML. Visual Studio [98] is utilized to implement the genetic algorithm.

5.2.2. Variables

The dependent variable in this experiment is model quality. The independent variables are security bad smells and quality metrics. During detection, the recall for the security bad smells in investigated use case diagrams is the measure of the independent variable. For correction, the independent variable is correction efficacy in terms of removal percentage of security bad smells. The description of instances of security bad smells in investigated use case diagrams is already provided in section 5.2.1. The other type of independent variable is quality metrics. This type is for the purpose of quantitative validation of security improvement. The quality metrics selected for use case diagram are as follows:

- *isExtension* identifies if a given use case(u) is an extension use case or not.
- *numAssMetric* is to count the number of association(s) between a use case and actor(s).
- *extendingMetric* is calculating a number of use cases, the extension use case is extending.
- *isInclusion* means if a use case is included by another use case or not.
- *includedMetric* counts the number of use cases, an inclusion use case is included by.

5.2.3. Proposed Hypotheses

The hypotheses are formulated to statistically address the posed research questions. Following hypotheses are formulated to statistically validate the effectiveness of our proposed approaches and make statistical judgment on security improvement in use case diagrams:

Hypothesis 1 (RQ1): The proposed detection technique is able to identify a significant number of security bad smells in the investigated use case diagrams.

Null Hypothesis (H_{01}): The detection approach is unable to identify a significant number of security bad smells in the investigated use case diagrams as indicated by its recall.

Alternate Hypothesis (H_{11}): The detection approach is able to identify a significant number of security bad smells in the investigated use case diagrams as indicated by its recall.

The null hypothesis (H_{01}) is rejected in the case, where, the Detection Recall (DR) of detection technique in terms of identifying the security bad smells in the investigated use case diagrams is significant. The quantification of formulated hypothesis is necessary for later testing. The quantification of our hypothesis is presented below in terms of detection recall:

Null Hypothesis (H_{01}): DR < 80%

Alternate Hypothesis (H_{11}): DR >= 80%

Hypothesis 2 (RQ2): The proposed correction technique is able to remove a significant number of security bad smells in the investigated use case diagrams.

Null Hypothesis (H_{02}): The correction approach is unable to remove a significant number of security bad smells in the investigated use case diagrams as indicated by its correction effectiveness.

Alternate Hypothesis (H_{12}): The correction approach is able to remove a significant number of security bad smells in the investigated use case diagrams as indicated by its correction effectiveness.

The null hypothesis (H_{02}) is rejected in the case, where, the Correction Efficacy (CE) of correction technique in terms of removing the security bad smells in the investigated use case diagrams is significant. The quantification of formulated hypothesis is necessary for later testing. The quantification of our hypothesis is presented below in terms of correction efficacy:

Null Hypothesis (H_{02}): CE < 80%

***Alternate Hypothesis* (*H_{12}*):** CE >= 80%

Hypothesis 3 (RQ3): Refactoring to security bad smells improves the investigated use case diagrams from a security perspective.

Null Hypothesis (H_{03}): No difference is observed in security quality of the investigated use case diagrams as a result of refactoring to security bad smells as indicated by quality metrics.

Alternate Hypothesis (H_{13}): A Significant difference is observed in security quality of the investigated use case diagrams as a result of refactoring to security bad smells as indicated by quality metrics.

The null hypothesis (H_{03}) is rejected in the case, where, quality metrics values before refactoring are not equal to quality metrics values after refactoring. The quantification of formulated hypothesis is necessary for later testing. The quantification of our hypothesis is presented below in terms of p-value:

Null Hypothesis (H_{03}): p-value > 0.05

***Alternate Hypothesis* (*H_{13}*):** p-value < 0.05

5.2.4. Experimental Tasks

Detection: The initial individuals are formed by governing rules from existing security bad smells in four use case diagrams. The aggregation of individuals creates initial population. The population undergoes selection, crossover and mutation operations as described in section 4.2.2. Once the genetic algorithm reaches its terminating condition, it yields a solution carrying best fitness. The selection of quality metrics and formation of the rules are accomplished through the measurement of metrics before and after refactoring. This allows us to identify the quality metrics which are affected by the refactoring to a specific bad smell, and later in the formulation of its rule.

Correction: The corrections in use case diagrams are accomplished by applying relevant refactoring techniques (mentioned in Table 3) to the identified security bad smells. The mapping of the listed anti-patterns to security bad smells are based on their descriptions and violations towards security aspects. The correction approach (described in Section 4.4) uses model transformation using XMI for refactoring purpose. The studied use case diagrams are exported to XML using Enterprise Architect. The relevant refactoring is applied by modifying/adding/deleting the tags in the XML representation. For example, in ATM system (Figure 7), there exists a security bad smell ‘missing hierarchy’, where actor ‘Bank’ is accessing ‘System Reporting’ extension use case. This smell is eradicated by removing the association between ‘Bank’ and ‘System Reporting’. The correction is made manually in XML representation of ATM system. The abridged version is shown in Figure 11. Only the tags affected from missing hierarchy are presented to ease the understandability. The first ‘links’ tag is carrying two ‘association’ tags, the second one represents the association between ‘Bank’ and ‘System Reporting’. This ‘Association’ tag needs to be removed from XML. The other ‘links’ tag contains ‘UseCase’ and ‘Association’ tags. This association needs to also be removed because it is adding to missing hierarchy problem. The rest are the ‘connector’ tags specifying the association’s source and target use case components. Since the missing hierarchy association has two components involved, so there are two connectors. In succession to other tag removals, both these connectors need to be removed to completely get rid of missing hierarchy security bad smell in ATM system. The other security bad smells are removed using similar process through related refactoring techniques specified in Table 3.

```
<links>
<Association
xmi:id="EAID_02E0767F_B82B_4356_B55C_CA5FB4A2710A"start="EAID_0CE06897_6B0D_449c_87
18_113528092FD6" end="EAID_2016AA29_DDF7_4c6d_9504_06A32D5B51EC"/>
<Association
xmi:id="EAID_6D15C31C_D6BF_4f70_A445_D453AA6FC964"start="EAID_0CE06897_6B0D_449c_8
718_113528092FD6" end="EAID_E4BBC282_8498_47f7_AF46_36B965BB5007"/>
</links>
<links>
<UseCase
xmi:id="EAID_122D424B_081A_4c7c_A30F_F16D6EBE6080"start="EAID_2016AA29_DDF7_4c6d_95
04_06A32D5B51EC" end="EAID_32ACC616_C186_4ca9_A9E6_027D300B43F3"/>
<Association
xmi:id="EAID_02E0767F_B82B_4356_B55C_CA5FB4A2710A"start="EAID_0CE06897_6B0D_449c_87
18_113528092FD6" end="EAID_2016AA29_DDF7_4c6d_9504_06A32D5B51EC"/>
</links>
<connector xmi:idref="EAID_02E0767F_B82B_4356_B55C_CA5FB4A2710A">
<source xmi:idref="EAID_0CE06897_6B0D_449c_8718_113528092FD6">
        <model ea_localid="49" type="Actor" name="Bank"/>
        <role visibility="Public" targetScope="instance"/>
        <type aggregation="none" containment="Unspecified"/>
        <constraints/>
        <modifiers isOrdered="false" changeable="none" isNavigable="false"/>
        <style value="Union=0; Derived=0; AllowDuplicates=0; Owned=0; Navigable=Unspecified;"/>
        <documentation/>
        <xrefs/>
        <tags/>
</source>
<target xmi:idref="EAID_2016AA29_DDF7_4c6d_9504_06A32D5B51EC">
        <model ea_localid="51" type="UseCase" name="System Reporting"/>
        <role visibility="Public" targetScope="instance"/>
        <type aggregation="none" containment="Unspecified"/>
        <constraints/>
        <modifiers isOrdered="false" changeable="none" isNavigable="true"/>
        <style value="Union=0; Derived=0; AllowDuplicates=0; Owned=0; Navigable=Navigable;"/>
        <documentation/>
        <xrefs/>
        <tags/>
</target>
        <model ea_localid="43"/>
        <properties ea_type="Association" direction="Source -&gt; Destination"/>
        <modifiers isRoot="false" isLeaf="false"/>
        <parameterSubstitutions/>
        <documentation/>
```

```
            <appearance linemode="3" linecolor="-1" linewidth="0" seqno="0" headStyle="0" lineStyle="0"/>
            <labels/>
            <extendedProperties virtualInheritance="0"/>
            <style/>
            <xrefs/>
            <tags/>
</connector>
<connector xmi:idref="EAID_122D424B_081A_4c7c_A30F_F16D6EBE6080">
<source xmi:idref="EAID_2016AA29_DDF7_4c6d_9504_06A32D5B51EC">
            <model ea_localid="51" type="UseCase" name="System Reporting"/>
            <role visibility="Public" targetScope="instance"/>
            <type aggregation="none" containment="Unspecified"/>
            <constraints/>
            <modifiers isOrdered="false" changeable="none" isNavigable="false"/>
            <style value="Union=0; Derived=0; AllowDuplicates=0; Owned=0; Navigable=Non-Navigable;"/>
            <documentation/>
            <xrefs/>
            <tags/>
</source>
<target xmi:idref="EAID_32ACC616_C186_4ca9_A9E6_027D300B43F3">
            <model ea_localid="50" type="UseCase" name="System Maintenance"/>
            <role visibility="Public" targetScope="instance"/>
            <type aggregation="none" containment="Unspecified"/>
            <constraints/>
            <modifiers isOrdered="false" changeable="none" isNavigable="true"/>
            <style value="Union=0; Derived=0; AllowDuplicates=0; Owned=0; Navigable=Navigable;"/>
            <documentation/>
            <xrefs/>
            <tags/>
</target>
            <model ea_localid="32"/>
            <properties ea_type="UseCase" subtype="Extends" stereotype="extend" direction="Source -&gt;
Destination"/>
            <modifiers isRoot="false" isLeaf="false"/>
            <documentation/>
            <appearance linemode="3" linecolor="-1" linewidth="0" seqno="0" headStyle="0" lineStyle="0"/>
            <labels mb=" &#xA;«extend»"/>
            <extendedProperties conditional=" &#xA;«extend»" virtualInheritance="0"/>
            <style/>
            <xrefs/>
            <tags/>
</connector>
```

Figure 11. Abridged XML of ATM system

Behavioral consistency: The behavioral consistency of the refactored use case diagrams with the source use case diagrams is performed by checking post conditions. The presence of few post conditions is ensured for each security bad smell. The problem of missing hierarchy is itself an incorrect behavior instance, so the removal of it makes the use case diagram behaviorally sound. In some cases, where an actor is associated only with an extension use case, the refactoring makes the actor an unassociated entity in the use case diagram. Since, the actor is involved in the inappropriate execution of a functionality before refactoring, and becomes an unassociated actor, it can be removed as well. This does not impact the behavior of a use case diagram because an unassociated actor is not contributing to the diagram. The refactoring to broken modularization encapsulates the inclusion use case in the included use case. The post refactoring condition to validate behavioral consistency is the presence of inclusion functionality. Whenever included use case executes, the inclusion use case automatically executes. So, moving the functionality of inclusion use case to the included use case does not change the behavior. The new combined use case executes both the functionalities (included and inclusion) as one. The refactoring to missing modularization breaks the extension use case into the number of use cases which it extends. Before refactoring, the extension use case is extending multiple use cases, which in fact violates the exceptional behavioral confined for a use case. The breaking of exceptional functionalities for each extended use case allows correct behavior. The refactoring does not only ensure the consistency but also it ensures the behavioral correctness.

5.2.5. Results

Detection: The GA yields a set of rules which represents the best solution. The solution generated by the execution of GA with use case diagrams is shown in Figure 12. R1 is measuring for missing hierarchy using two conditional statements having isExtension and numAssMetric variables. For example, in Figure 7, there exists an association between ‘System Reporting’ extension use case and ‘Bank’ actor, which is an instance of missing hierarchy security bad smell. Similarly, R2 is focusing on detecting missing modularization bad smell. This rule also uses two conditional statements with variables: isExtension and extendingMetric. For example, in Figure 7, ‘Bad Pin’ use case is extending two use cases, which is a clear missing modularization problem. Lastly, R3 is concentrating on broken modularization. It uses two conditional statements having variables: isInclusion and includedMetric. For example, in Figure 7, ‘System Shutdown’ use case is included by a use case. This should be refactored because it is causing broken modularization problem.

R1: IF (isExtension (u) == true AND numAssMetric (u) >= 1) THEN missing hierarchy

R2: IF (isExtension (u) == true AND extendingMetric (u) >= 2) THEN missing modularization

R3: IF (isInclusion (u) == true AND includedMetric (u) == 1) THEN broken modularization

Figure 12. Best solution generated for use case diagrams

The best set of rules is then applied on investigated use case diagrams to evaluate its recall efficiency. The set of rules governing best solution are able to identify all, 27, security bad smells present in examined four use case diagrams, meaning, detection approach has 100% recall. To further confirm this, the detected smells are also validated manually.

Correction: The correction procedure along with the instance of ATM system is provided in section 5.2.4. The Same procedure is applied to remove security bad smells in other investigated use case diagrams. Our correction technique is able to remove all security bad smells in investigated use case diagrams. We are presenting the details about how each refactoring is applied to the corresponding security bad smell in each investigated use case diagram in the form of tables. Table 10, 11, 12 and 13 summarizes the refactoring application to security bad smells in ATM System, Human Resource System, Restaurant System and Travel Agency System respectively. The resulting use case diagrams of ATM

System, Human Resource System, Restaurant System and Travel Agency System after refactoring application are depicted in Figure 13, 14, 15, 16 respectively. It can be observed from the refactored diagrams that the identified security bad smells are removed.

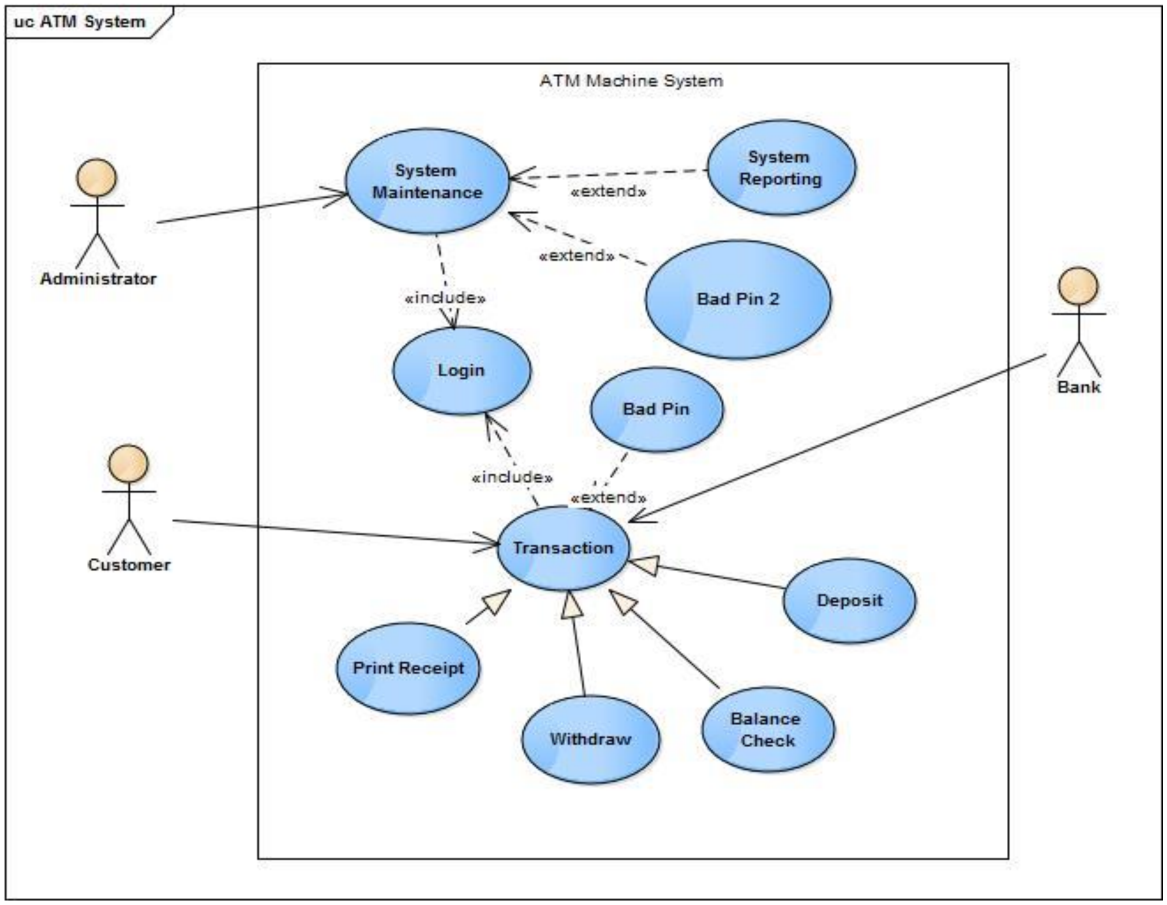

Figure 13. Refactored use case diagram of ATM system

Table 10. Applied refactoring in ATM system

Security bad smell ID	Applied refactoring
UB1	Drop association between Bank and System Reporting
UB2	Split Bad Pin extension use case into two extension use cases
UB3	Drop System Shutdown inclusion use case

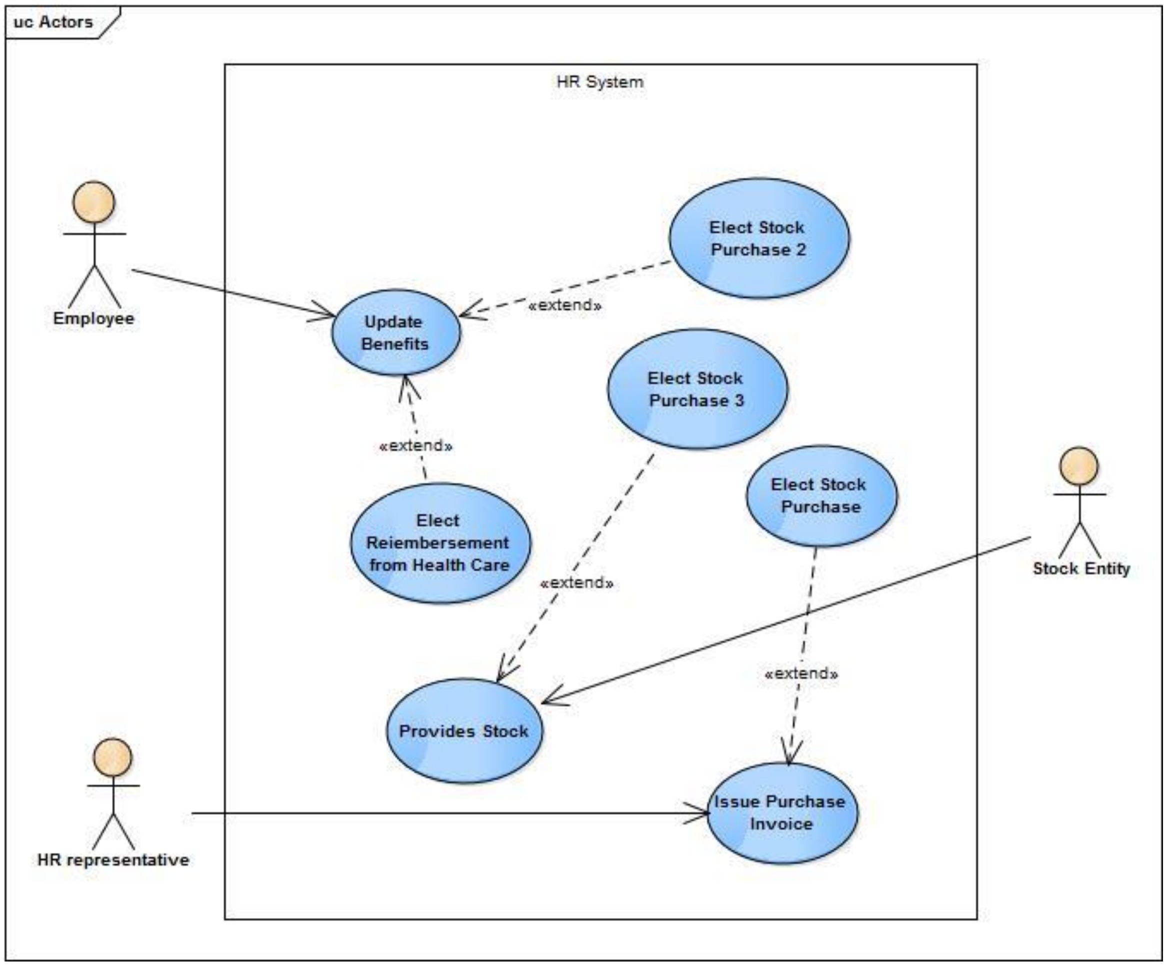

Figure 14. Refactored use case diagram of HR system

Table 11. Applied refactoring in HR system

Security bad smell ID	**Applied refactoring**
UB4	Drop association between health Care Dept. and Elect Reimbursement from Health Care
UB5	Drop association between Stock Entity and Elect Stock Purchase
UB6	Split Elect Stock Purchase extension use case into three extension use cases
UB7	Drop Update Dental Plan inclusion use case
UB8	Drop Update Insurance Plan inclusion use case
UB9	Drop Update Medical Plan inclusion use case

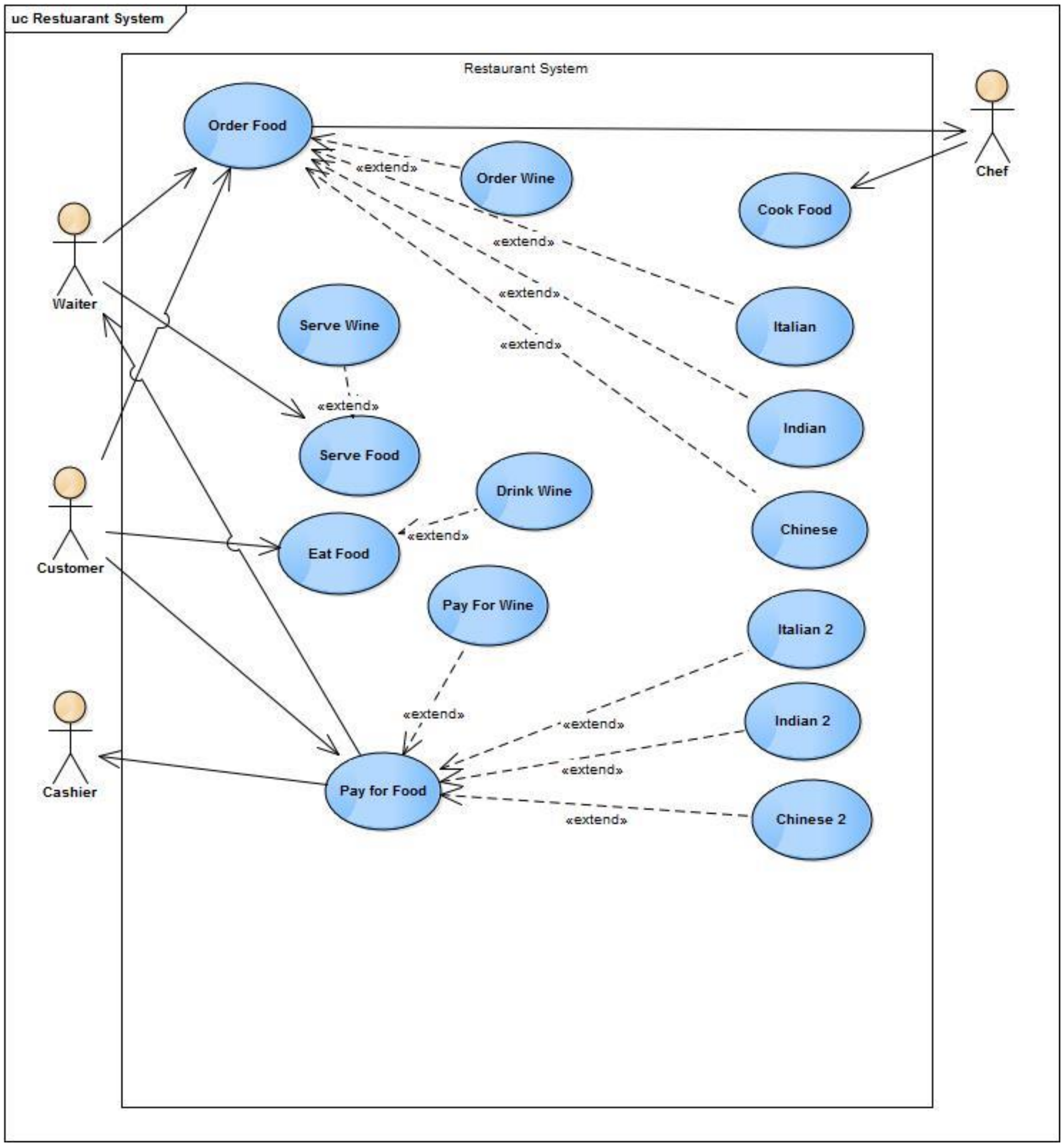

Figure 15. Refactored use case diagram of restaurant system

Table 12. Applied refactoring in restaurant system

Security bad smell ID	**Applied refactoring**
UB10	Drop association between Waiter and Order Wine
UB11	Drop association between Customer and Order Wine
UB12	Drop association between Chef and Order Wine
UB13	Drop association between Waiter and Serve Wine
UB14	Drop association between Customer and Drink Wine
UB15	Drop association between Customer and Pay for Wine
UB16	Drop association between Waiter and Pay for Wine
UB17	Split Chinese extension use case into two extension use cases
UB18	Split Italian extension use case into two extension use cases
UB19	Split Indian extension use case into two extension use cases
UB20	Drop Pay Tip inclusion use case

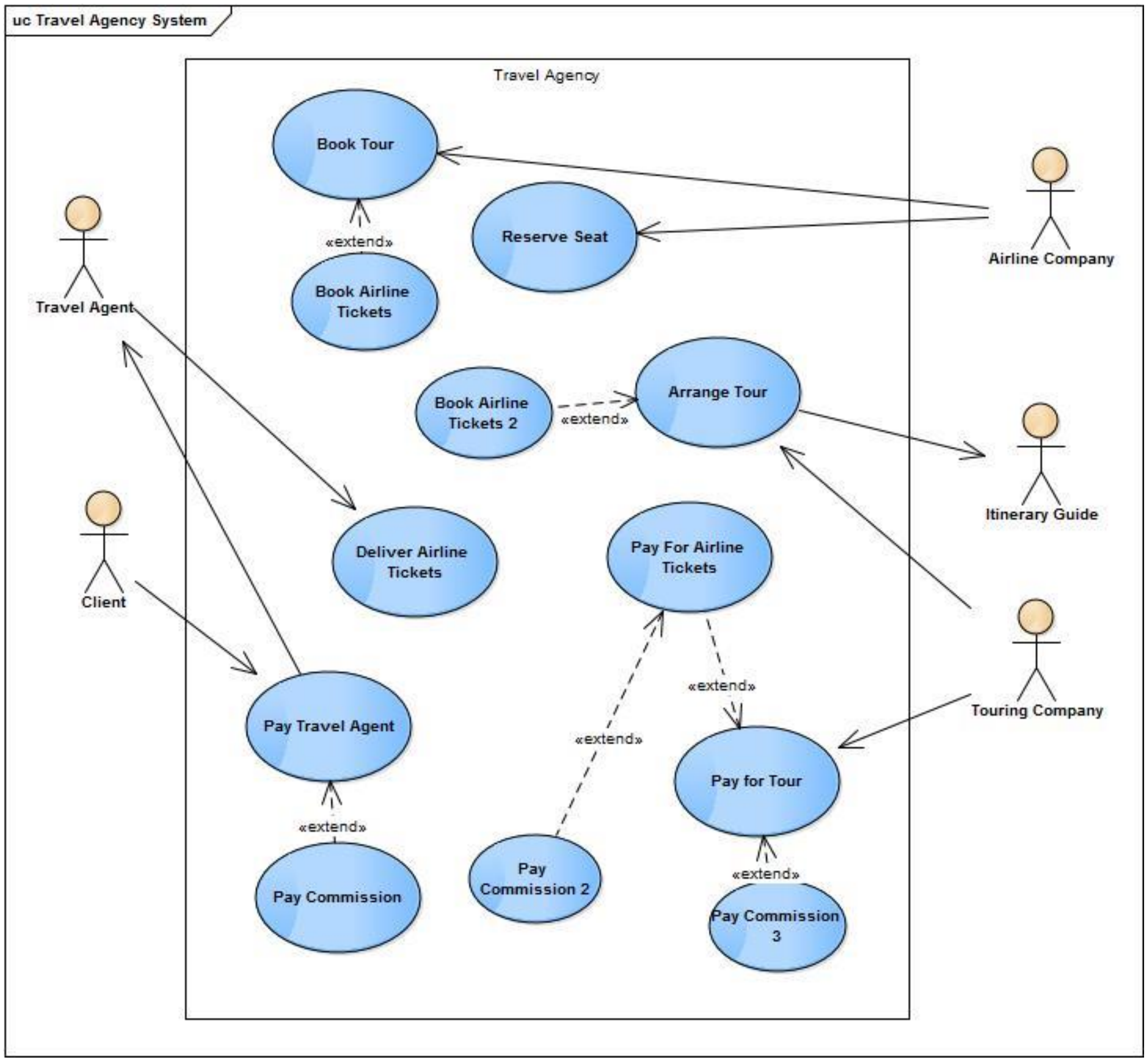

Figure 16. Refactored use case diagram of travel agency system

Table 13. Applied refactoring in travel agency system

Security bad smell ID	Applied refactoring
UB21	Drop association between Travel Agent and Book Airline Tickets
UB22	Drop association between Airline Company and Book Airline Tickets
UB23	Drop association between Client and Book Airline Tickets
UB24	Drop association between Airline Company and Pay for Airline Tickets
UB25	Split Book Airline Tickets extension use case into two extension use cases
UB26	Split Pay Commission extension use case into three extension use cases
UB27	Drop Pay for Airline Tickets inclusion use case

5.2.6. Hypotheses Testing

To reiterate, we have formulated three hypothesis addressing our three research questions. Each hypothesis is numerically validated as follows:

Hypothesis 1 (RQ1): In order to test this hypothesis, the Detection Recall (DR) of our detection approach is measured. The null hypothesis (H_{01}) can be rejected, if DR is significant. Numerically, it is set that if DR is greater than or equal to 80%, then null hypothesis (H_{01}) can be rejected. Our detection approach shows a significant DR of 100% while executing on investigated use case diagrams. The DR is greater than 80%, so null hypothesis (H_{01}) is rejected. This answers our RQ1 that our proposed detection approach is able to detect significant number of the security bad smells in use case diagrams.

Hypothesis 2 (RQ2): In order to test this hypothesis, the Correction Efficacy (CE) of our correction approach is measured. The null hypothesis (H_{02}) can be rejected, if CE is significant. In numerical terms, if CE is greater than or equal to 80%, the null hypothesis (H_{02}) can be rejected. Our correction approach shows overwhelming results by yielding a significant CE of 100% in investigated use case diagrams. The CE is greater than 80%, so null hypothesis (H_{01}) is rejected. This addresses our RQ2 that our proposed correction approach is able to remove significant number of the security bad smells in use case diagrams.

Hypothesis 3 (RQ3): Statistical analysis is performed to conclude on the contribution of refactoring in security improvement of use case diagrams. To statistically analyze whether security has significantly improved in use case diagrams, we apply pair-wise t-test. The pair-wise t-test is beneficial in this case as it would be able to identify the differences in quality metrics as a result of refactoring. The p-value is computed with 95% confidence using the pair-wise t-test. It is noticed that the computed p-value is 0.0001, which is less than 0.05. This justifies significant security improvement in investigated use case diagrams and subsequently, answers our RQ3. In succession to this observation, we can reject our formulated null hypothesis (H_{03}) with 95% confidence. By rejecting the null hypothesis, the sub-goal of security improvement in use case diagrams is achieved. For reference, the quality metrics values pre and post refactoring are provided in Appendix D.

5.3. Experimental Design for Sequence Diagrams

This section aims to provide details on our experiment with sequence diagrams. The following subsections describe our experiment in terms of experimental materials, variables, proposed hypotheses, experimental tasks, results and hypotheses testing.

5.3.1. Experimental Materials

Five sequence diagrams belonging to five different systems are used for the detection purpose. The diagrams are gathered from an online source [96]. The selection of the sequence diagrams is achieved through random sampling to avoid any biasness towards our results. As mentioned earlier in Table 8, the investigated security bad smells in sequence diagram are missing modularization, broken modularization and unutilized abstraction. Multiple instances (total 20) of these three security bad smells can be seen in investigated sequence diagrams. The multiple instances of same bad smell allow diversity in generated rules, which contributes to solution effectiveness.

Figure 17 shows the sequence diagram of airline reservation system [96]. The diagram comprises of five classes having associations among them except ‘Reservation System’ class. This class is assumed to be an unutilized abstract class. The presented airline reservation system carries three security bad smells: missing modularization, broken modularization and unutilized abstraction. The instances where these bad smells occur are listed as follows:

Missing modularization:

SB1: ‘Customer’ class has a lot of associations and in-out calls or messages.

Broken modularization:

SB2: ‘Flight’ class has just one received call.

Unutilized abstraction:

SB3: ‘Reservation System’ is unassociated with any other class.

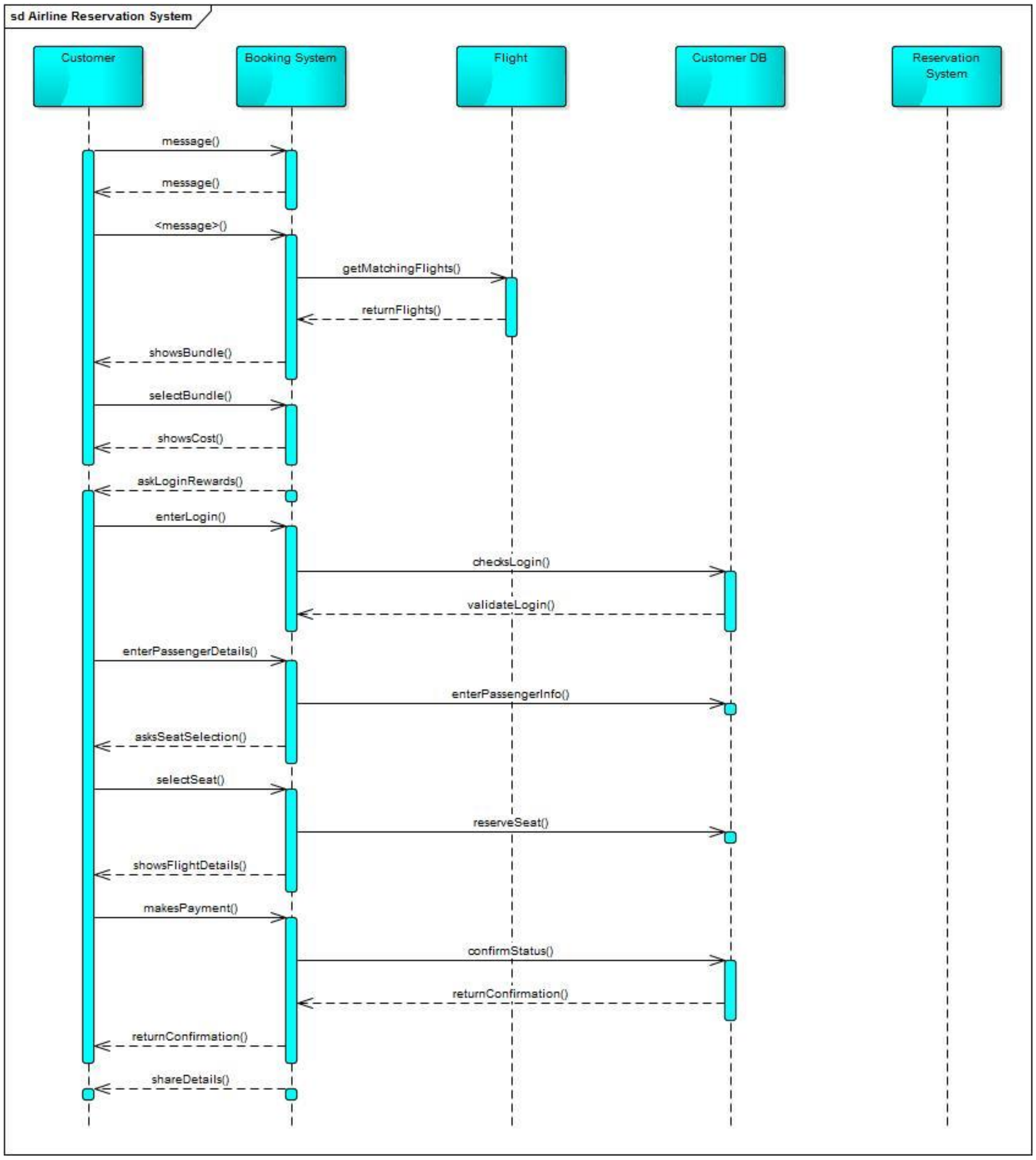

Figure 17. Sequence diagram of airline reservation system

Figure 18 shows the sequence diagram of hotel management system [96]. The diagram comprises of nine classes having associations among them except ‘Staff’ class. This class is assumed to be an unutilized abstract class. The presented hotel management system carries three security bad smells: missing modularization, broken modularization and unutilized abstraction. The instances where these bad smells occur are listed as follows:

Missing modularization:

SB4: ‘Receptionist’ class has a lot of associations and in-out calls or messages.

SB5: ‘Customer’ class has a lot of associations and in-out calls or messages.

Broken modularization:

SB6: ‘Stock’ class has just one received call.

SB7: ‘Food Items’ class has just one received call.

SB8: ‘Room Attendant’ class has just one received call.

Unutilized abstraction:

SB9: ‘Staff’ is unassociated with any other class.

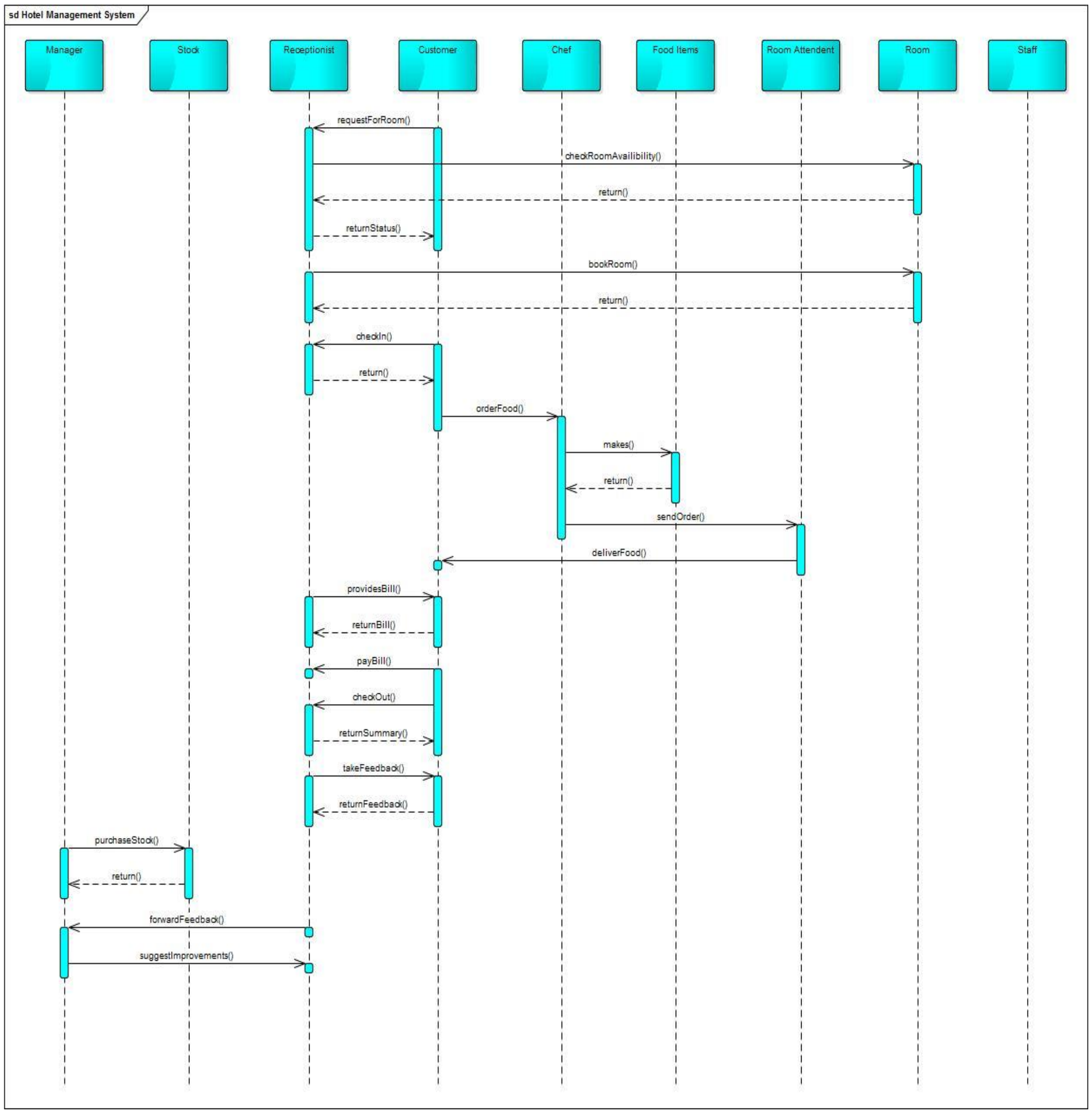

Figure 18. Sequence diagram of hotel management system

Figure 19 shows the sequence diagram of library management system [96]. The diagram comprises of six classes having associations among them except ‘Staff’ class. This class is assumed to be an unutilized abstract class. The presented library management system carries three security bad smells: missing modularization, broken modularization and unutilized abstraction. The instances where these bad smells occur are listed as follows:

Missing modularization:

SB10: ‘Librarian’ class has a lot of associations and in-out calls or messages.

SB11: ‘User’ class has a lot of associations and in-out calls or messages.

Broken modularization:

SB12: ‘Manager’ class has just one received call.

Unutilized abstraction:

SB13: ‘Staff’ class is unassociated with any other class.

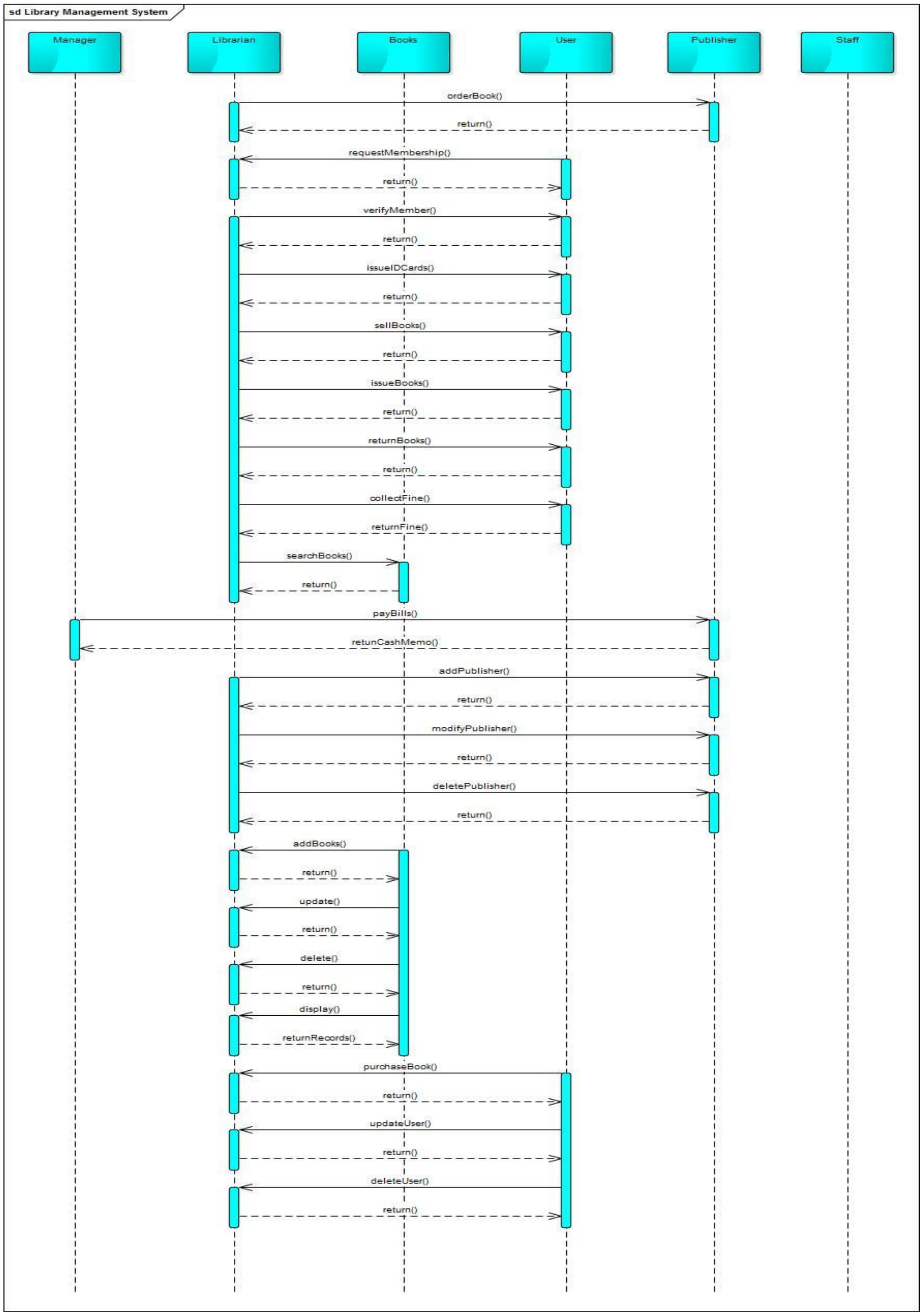

Figure 19. Sequence diagram of library management system

Figure 20 shows the sequence diagram of online movie ticketing system [96]. The diagram comprises of eight classes having associations among them except ‘Visitor’ and ‘Ticket’ classes. These classes are assumed to be unutilized abstract classes. The presented online movie ticketing system carries three security bad smells: missing modularization, broken modularization and unutilized abstraction. The instances where these bad smells occur are listed as follows:

Missing modularization:

SB14: ‘Registered User’ class has a lot of associations and in-out calls or messages.

Broken modularization:

SB15: ‘Cancel Ticket’ class has just one received call.

Unutilized abstraction:

SB16: ‘Visitor’ class is unassociated with any other class.

SB17: ‘Ticket’ class is unassociated with any other class.

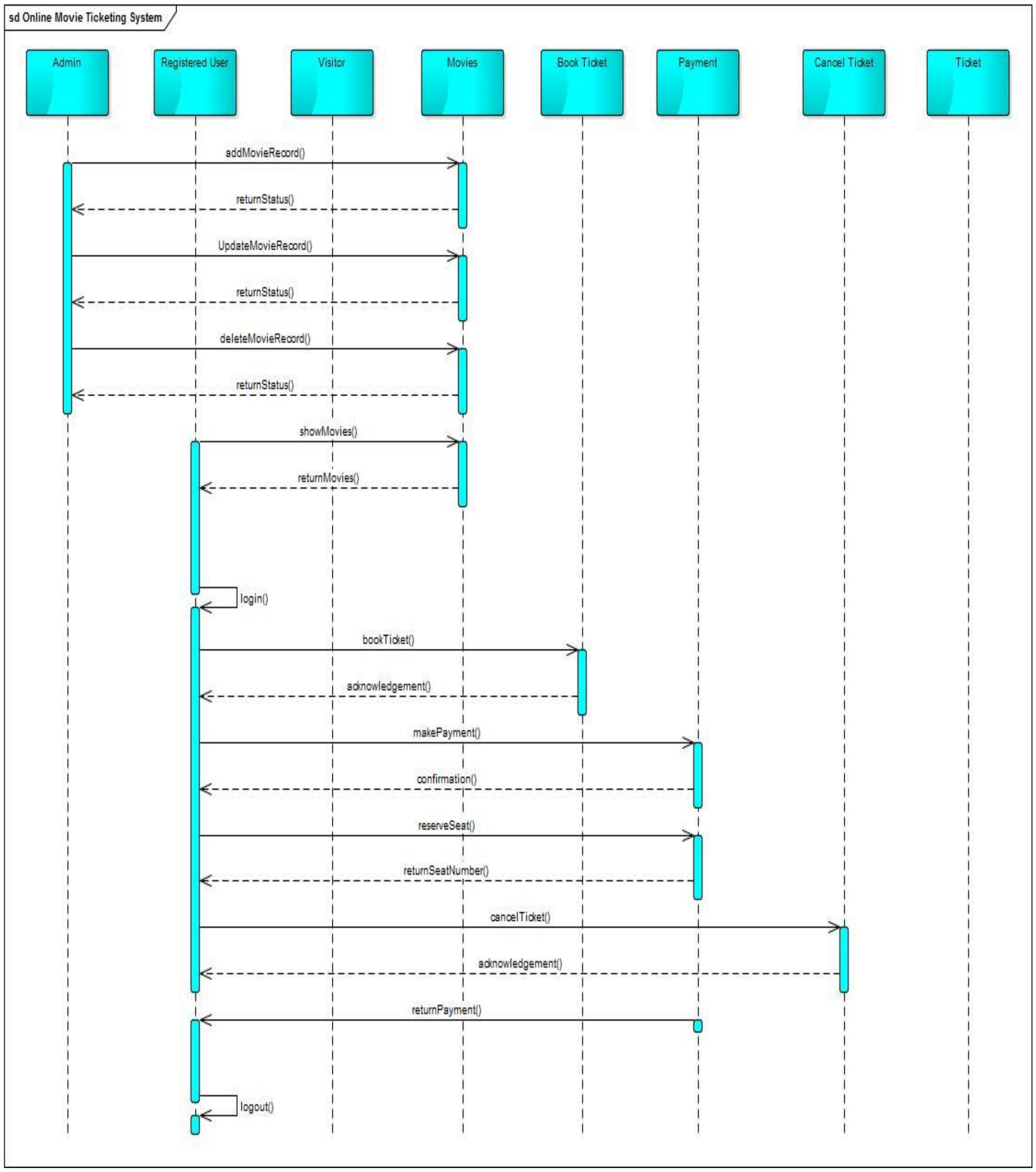

Figure 20. Sequence diagram of online movie ticketing system

Figure 21 shows the sequence diagram of school management system [96]. The diagram comprises of five classes having associations among them except ‘Employee’ class. This class is assumed to be an unutilized abstract class. The presented school management systems system carries three security bad smells: missing modularization, broken modularization and unutilized abstraction. The instances where these bad smells occur are listed as follows:

Missing modularization:

SB18: ‘Admin’ class has a lot of associations and in-out calls or messages.

Broken modularization:

SB19: ‘Class’ class has just one received call.

Unutilized abstraction:

SB20: ‘Employee’ class is unassociated with any other class.

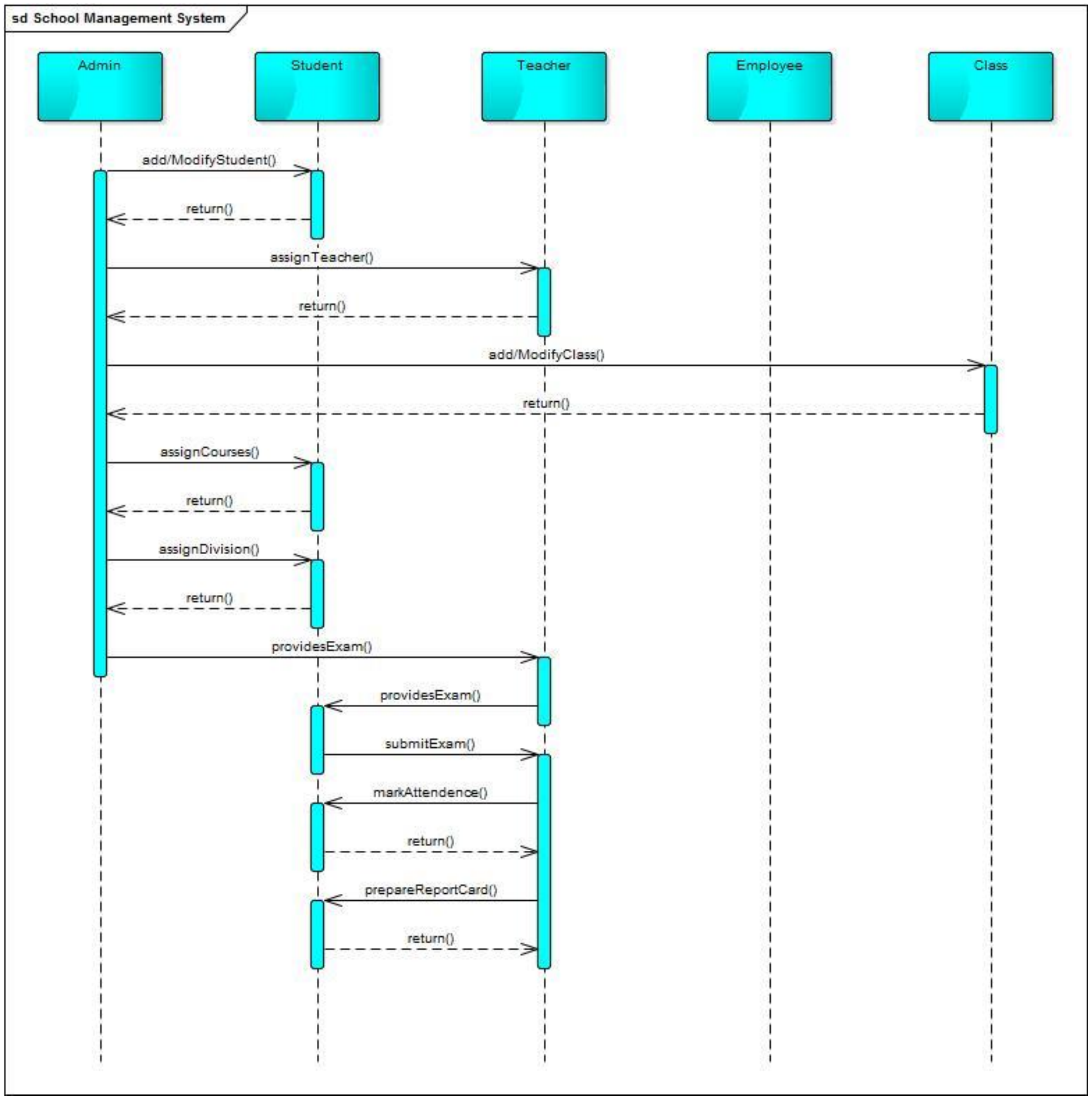

Figure 21. Sequence diagram of school management system

Although SDMetrics tool provides the metrics values of sequence diagrams, the provided set of metrics does not contribute to the investigated security bad smells in our research. So quality metrics in sequence diagrams are calculated manually. The other tools such as Visual Studio and Enterprise Architect are used in the same manner and for the same purposes as for use case diagrams.

5.3.2. Variables

The dependent and independent variables are the same as described in section 5.2.2. The independent variable for detection of security bad smells in investigated sequence diagrams is detection recall. For correction, the independent variable is computed as how many security bad smells are removed as a result of refactoring. The description of instances of security bad smells in investigated sequence diagrams is already provided in section 5.3.1. The other independent variable is quality metrics. The metrics are useful in the quantitative validation of security improvement. The quality metrics selected for sequence diagram are as follows:

- *NAss* is the number of in-out messages or calls, a class exhibits.
- *NInvoc* is the number of invoked calls of a class.
- *NRec* is the number of received messages for a class.
- *CBO* is the number of coupled classes with a class.

5.3.3. Proposed Hypotheses

Following hypotheses are formulated to statistically validate the effectiveness of our proposed approaches and make statistical judgment on security improvement in sequence diagrams:

Hypothesis 4 (RQ1): The proposed detection technique is able to identify a significant number of security bad smells in the investigated sequence diagrams.

Null Hypothesis (H_{04}): The detection approach is unable to identify a significant number of security bad smells in the investigated sequence diagrams as indicated by its recall.

Alternate Hypothesis (H_{14}): The detection approach is able to identify a significant number of security bad smells in the investigated sequence diagrams as indicated by its recall.

The null hypothesis (H_{04}) is rejected in the case, where, the Detection Recall (DR) of detection technique in terms of identifying the security bad smells in investigated sequence diagrams is significant. The quantification of formulated hypothesis is necessary for later testing. The quantification of our hypothesis is presented below in terms of detection recall:

Null Hypothesis (H_{04}): DR < 80%

Alternate Hypothesis (H_{14}): DR >= 80%

Hypothesis 5 (RQ2): The proposed correction technique is able to remove a significant number of security bad smells in the investigated sequence diagrams.

Null Hypothesis (H_{05}): The correction approach is unable to remove a significant number of security bad smells in the investigated sequence diagrams as indicated by its correction effectiveness.

Alternate Hypothesis (H_{15}): The correction approach is able to remove a significant number of security bad smells in the investigated sequence diagrams as indicated by its correction effectiveness.

The null hypothesis (H_{05}) is rejected in the case, where, the Correction Efficacy (CE) of correction technique in terms of removing the security bad smells in investigated sequence diagrams is significant. The quantification of formulated hypothesis is necessary for later testing. The quantification of our hypothesis is presented below in terms of correction efficacy:

Null Hypothesis (H_{05}): CE < 80%

***Alternate Hypothesis* (*H_{15}*):** CE >= 80%

Hypothesis 6 (RQ3): Refactoring to security bad smells improves the investigated sequence diagrams from a security perspective.

Null Hypothesis (H_{06}): No difference is observed in security quality of the investigated sequence diagrams as a result of refactoring to security bad smells as indicated by quality metrics.

Alternate Hypothesis (H_{16}): A Significant difference is observed in security quality of the investigated sequence diagrams as a result of refactoring to security bad smells as indicated by quality metrics.

The null hypothesis (H_{06}) is rejected in the case, where, quality metrics values before refactoring are not equal to quality metrics values after refactoring. The quantification of formulated hypothesis is necessary for later testing. The quantification of our hypothesis is presented below in terms of p-value:

Null Hypothesis (H_{06}): p-value > 0.05

***Alternate Hypothesis* (*H_{16}*):** p-value < 0.05

5.3.4. Experimental Tasks

Detection: The similar detection process is applied for sequence diagrams as is for use case diagrams. The initial individuals are formed by governing rules from existing security bad smells in five sequence diagrams. The aggregation of individuals creates initial population. The population undergoes selection, crossover and mutation operations as described in section 4.2.2. Once genetic algorithm reaches its terminating condition, it yields a solution carrying best fitness.

Correction: The corrections in sequence diagrams are achieved by applying relevant refactoring techniques (mentioned in Appendix C) to identified security bad smells. Once again, same model transformation procedure (described in Section 4.4) is applied for correction of sequence diagrams. The investigated sequence diagrams are exported using XML to cater refactoring by modifying the XML representation. For example, in Airline Reservation System (Figure 17), there exists a security bad smell ‘unutilized abstraction’, where ‘Reservation System’ abstract class is not utilized at all. This smell is eradicated by removing the ‘Reservation System’ abstract class from sequence diagram. This system is first exported to XML representation and then the tags related to this abstract class are removed. The other security bad smells are removed using the same process through related refactoring techniques specified in Appendix C.

Behavioral consistency: Once again, for sequence diagram, the behavioral consistency is fulfilled using post refactoring conditions. The unutilized abstraction does not contribute to the sequence diagram, so the refactoring to this bad smell does not introduce any behavioral consistency issue. The refactoring to broken modularization moves the method to the class which it needs. Previously, the class is calling it from another class and violates multiple security attributes. After refactoring, the functionality is moved to the class, which was calling it from another class, leaving the behavior untouched. The refactoring to missing modularization decomposes a class into two classes and distributes the relevant functionalities according to their concerns. The most important post refactoring condition to fulfill is the presence of all the functionalities after the decomposition. In this case, the semantics are present and requires the involvement of the designer. Another condition to fulfill is that the interactions of the decomposed class with other classes remain intact. In other words, the sending and receiving of messages between the refactored class and other classes should be present as they are supposed to be.

5.3.5. Results

Detection: Once the detection technique is applied on sequence diagrams, it generates the best solution. The set of rules, representing best solution, yielded by the genetic algorithm is shown in Figure 22. All three rules are measuring the bad smells by using four conditional statements having variables: NAss, NInvoc, NRec, and CBO. R1, R2 and R3 are measuring for missing modularization, broken modularization and unutilized abstraction respectively. The computation of quality metrics is performed manually due to unavailability of tools. The considered quality metrics and corresponding mapping of rules to specific bad smells are extracted from Fourati et.al [71]. If the metrics values of class(c) equal or exceeds the thresholds provided by these rules, then that class has a corresponding bad smell. The best solution (shown in Figure 22) is then applied on investigated sequence diagrams to evaluate its recall effectiveness. The set of rules governing best solution are able to identify 18 out of 20 security bad smells present in examined five sequence diagrams, meaning, detection approach has 90% recall. The two undetected security bad smells belong to missing modularization and are present in Airline Reservation System and Library Management System. The recall is validated manually as well to confirm the correct detection of security bad smells. The acquired recall provides sufficient evidence to fulfill our RQ1, that our proposed detection approach is able to detect a significant number of security bad smells in sequence diagrams.

R1: IF (NAss(c) >= 11 AND (NInvoc(c) >= 4 AND NRec(c) >= 4) AND CBO(c) >= 3) THEN missing modularization(c)
R2: IF (NAss(c) == 2 AND (NInvoc(c) == 1 OR NRec(c) == 1) AND CBO(c) == 1) THEN broken modularization(c)
R3: IF (NAss(c) == 0 AND NInvoc(c) == 0 AND NRec(c) == 0 AND CBO(c) == 0) THEN unutilized abstraction(c)

Figure 22. Best solution generated for sequence diagrams

Correction: The correction procedure along with the instance of ATM system is provided in section 5.2.4. The Same procedure is applied to remove security bad smells in investigated sequence diagrams. We are presenting the details about how each refactoring is applied to the corresponding security bad smell in each investigated sequence diagram in the form of tables. Table 14, 15, 16, 17 and 18 summarizes the refactoring application to identified security bad smells in sequence diagrams of Airline Reservation System, Hotel Management System, Library Management System, Online Movie Ticketing System and School Management System respectively. The sequence diagrams of Airline Reservation System, Hotel Management System, Library Management System, Online Movie Ticketing System and School Management System after refactoring application are shown in Figure 23, 24, 25, 26 and 27 respectively. 19 out of 20 security bad smells are eradicated through our correction approach, making 95% correction effectiveness.

Table 14. Applied refactoring in airline reservation system

Security bad smell ID	Applied refactoring
SB1	Extract Visitor class from Customer class and move relevant methods to it
SB2	Move method to Booking System and remove Flight class
SB3	Remove reservation system abstract class

Table 15. Applied refactoring in hotel management system

Security bad smell ID	Applied refactoring
SB4	Extract Assistant class from Receptionist class and move relevant methods to it
SB5	Extract Resident class from Customer class and move relevant methods to it
SB6	Move method to Manager class and remove Stock class
SB7	Move method to Chef class and remove Food Items class
SB8	Move method to Chef class and remove Room Attendant class
SB9	Remove Staff abstract class

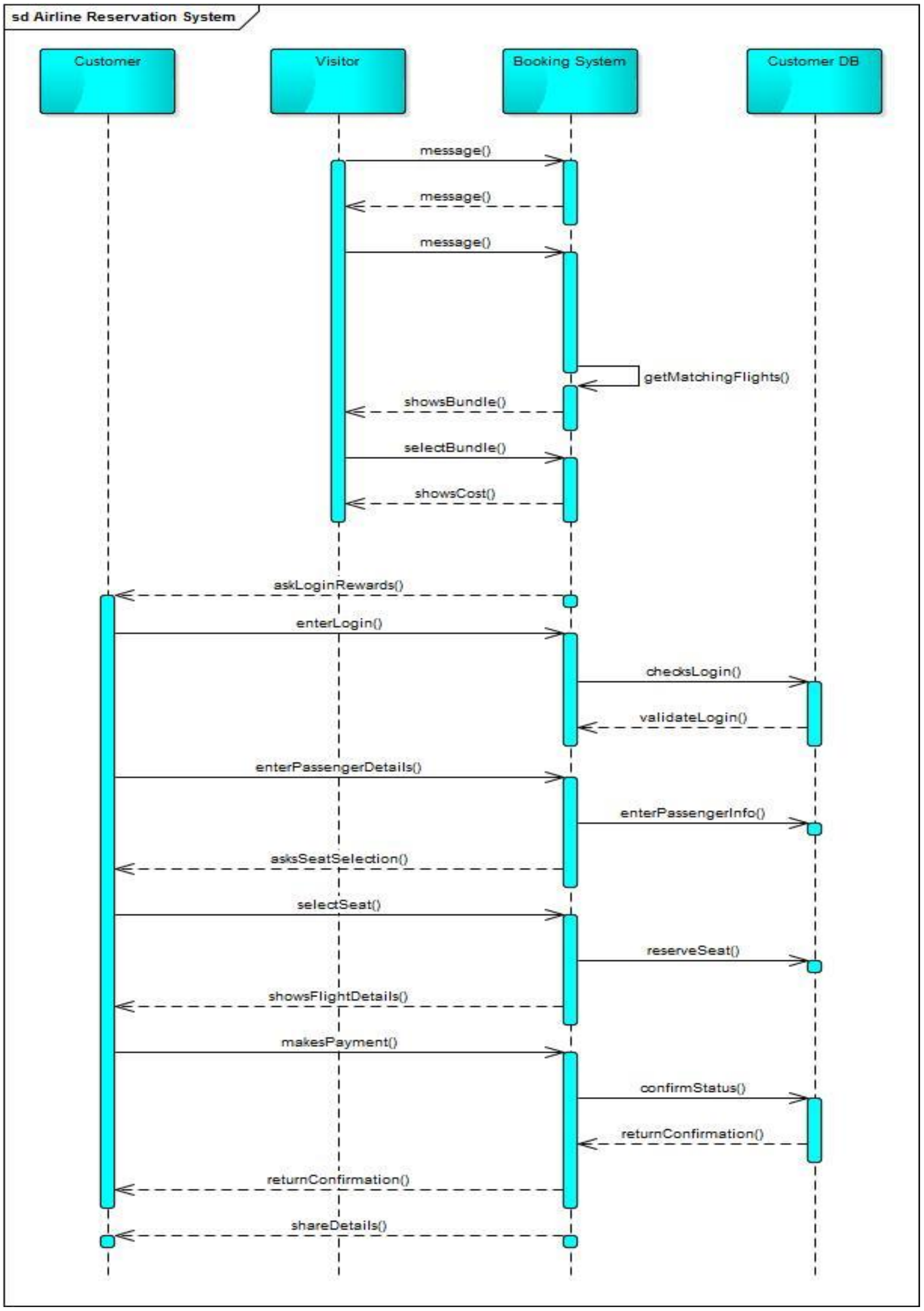

Figure 23. Refactored sequence diagram of airline reservation system

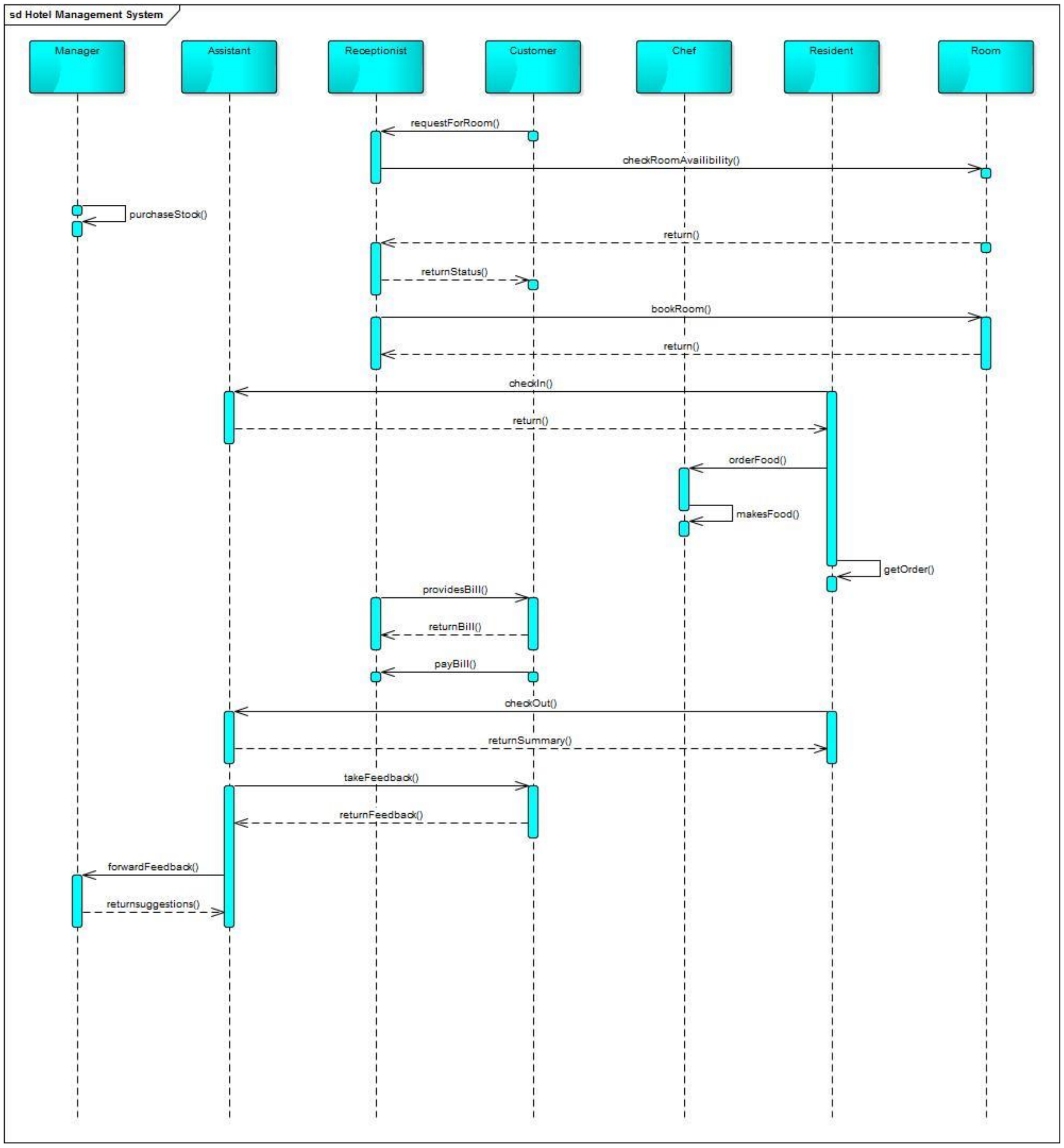

Figure 24. Refactored sequence diagram of hotel management system

Table 16. Applied refactoring in library management system

Security bad smell ID	**Applied refactoring**
SB10	Extract Assistant class from Librarian class and move relevant methods to it
SB11	Extract Premium User class from User class and move relevant methods to it
SB12	Move method to Publisher class and remove Manager class
SB13	Remove Staff abstract class

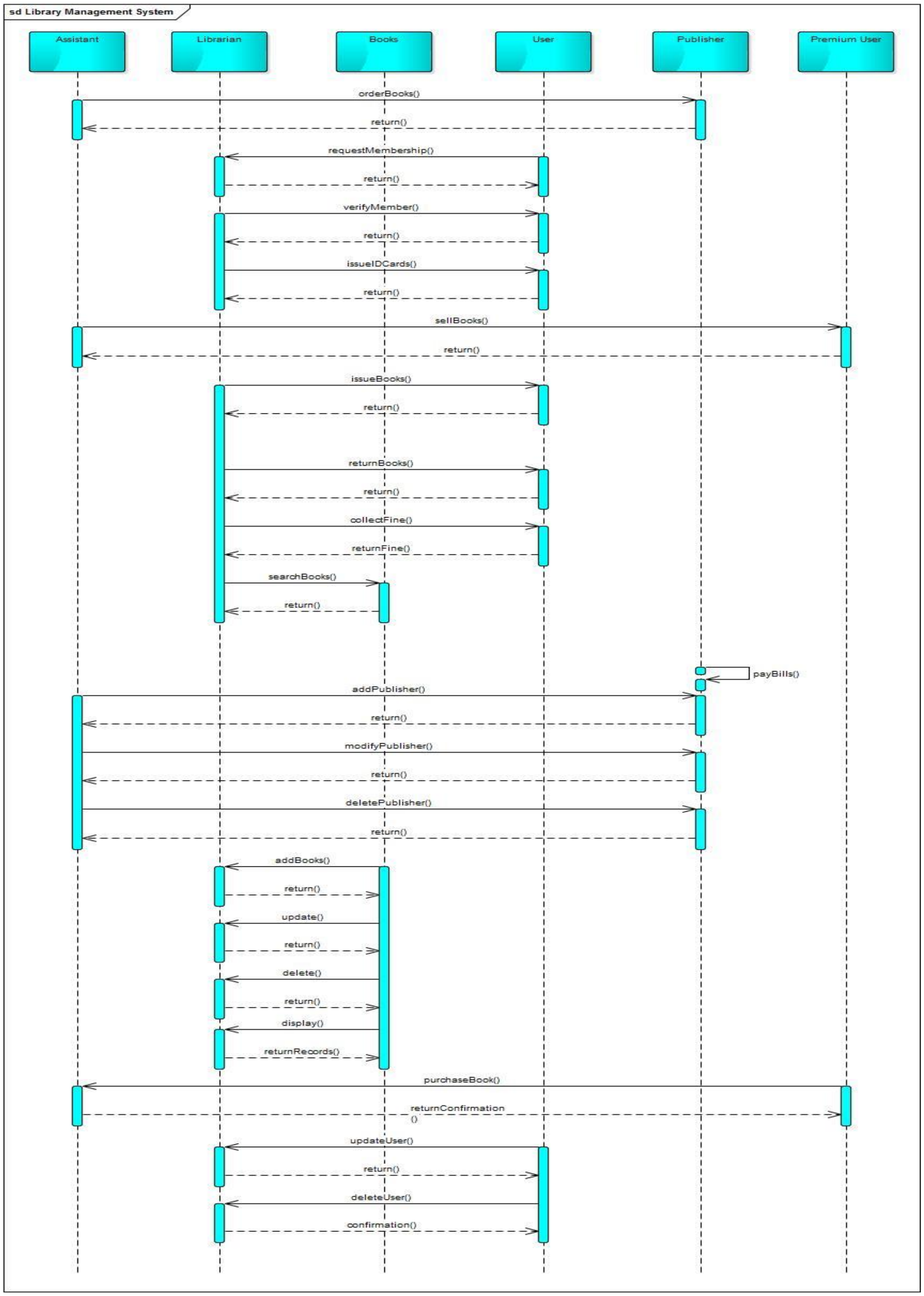

Figure 25. Refactored sequence diagram of library management system

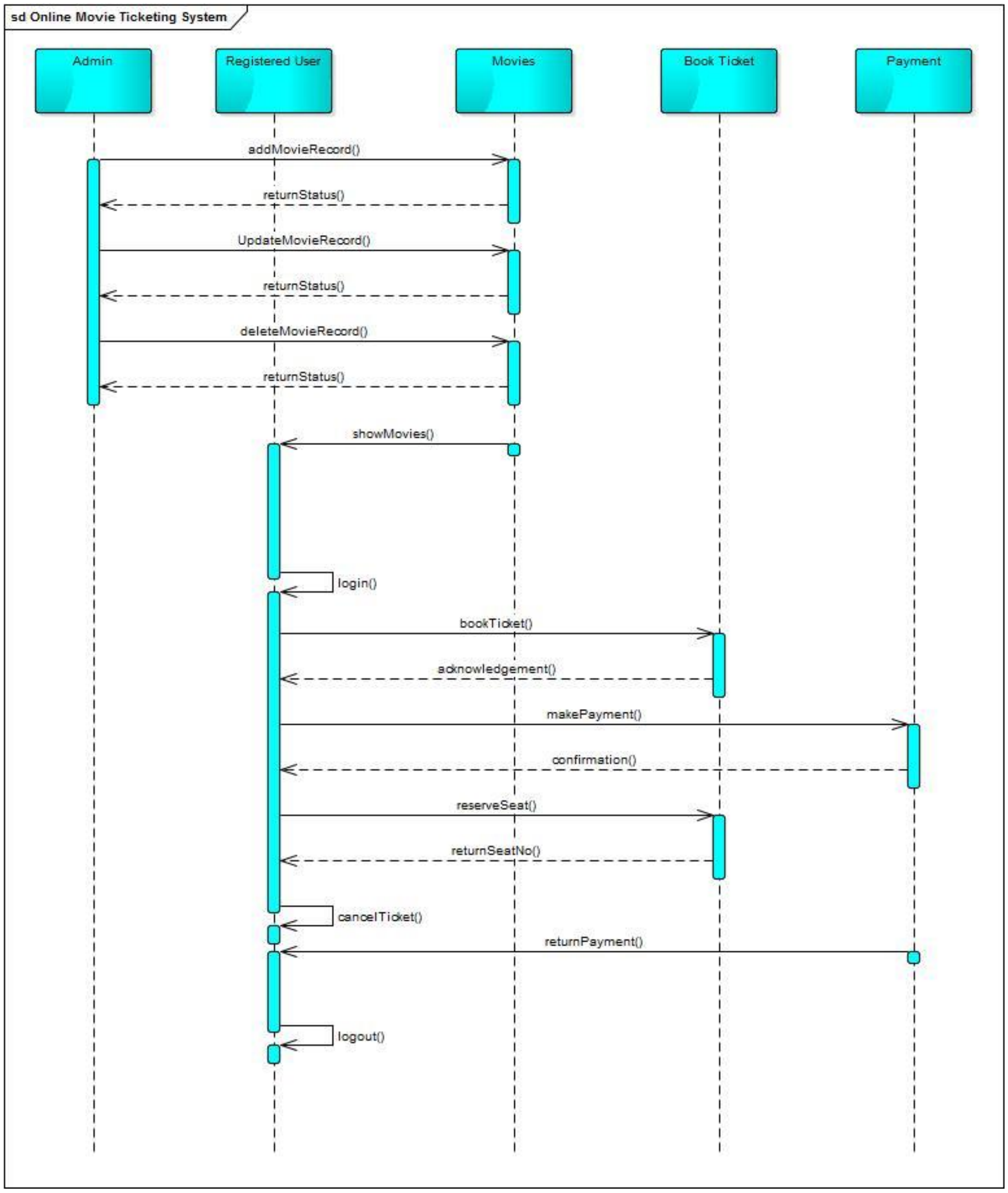

Figure 26. Refactored sequence diagram of online movie ticketing system

Table 17. Applied refactoring in online movie ticketing system

Security bad smell ID	**Applied refactoring**
SB14	The smell is automatically removed by refactoring other smells
SB15	Move method to Registered User class and remove Cancel Ticket class
SB16	Remove Visitor abstract class
SB17	Remove Ticket abstract class

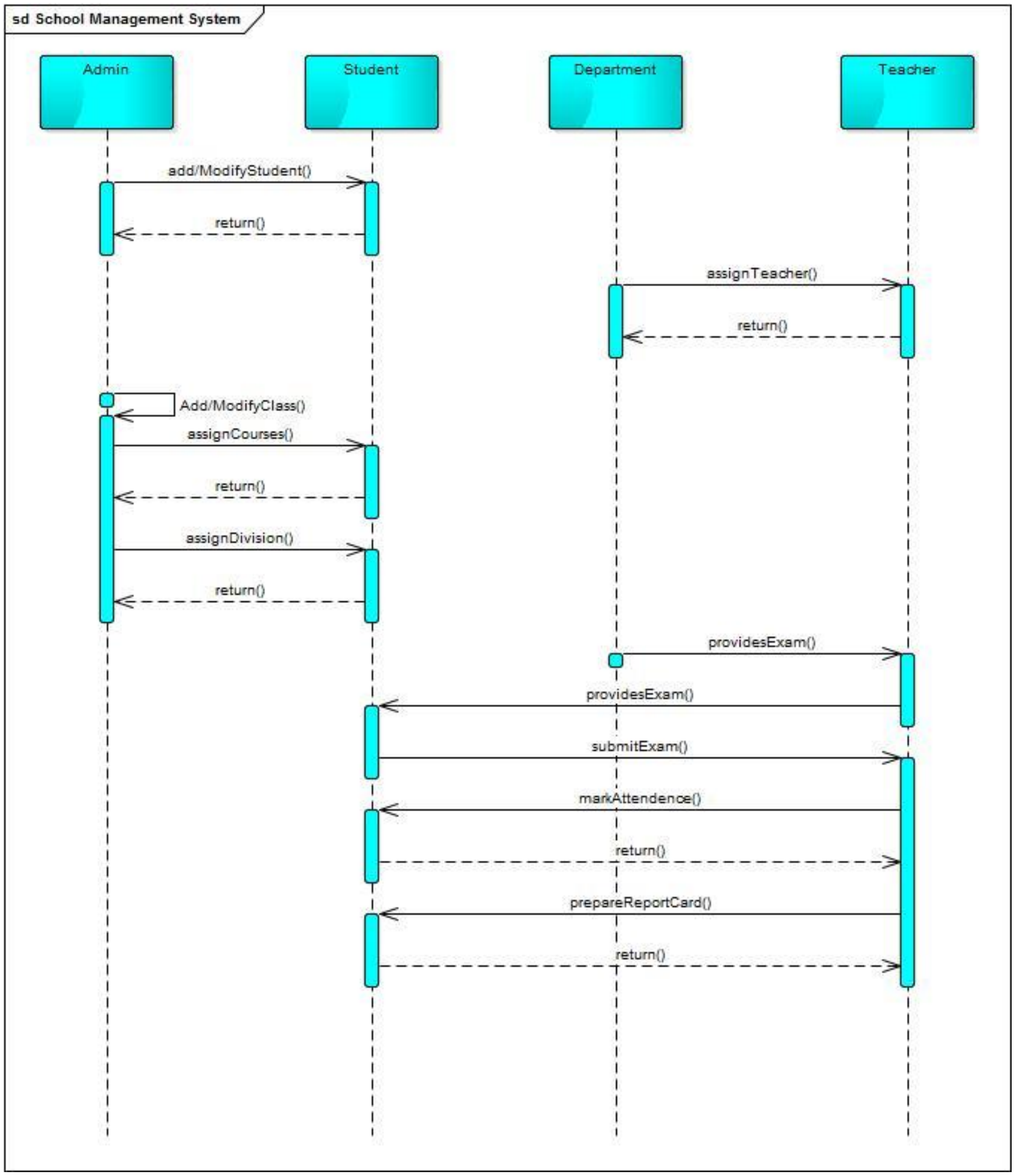

Figure 27. Refactored sequence diagram of school management system

Table 18. Applied refactoring in school management system

Security bad smell ID	**Applied refactoring**
SB18	Extract Department class from Admin class and move relevant methods to it
SB19	Move method to Admin class and remove Class class
SB20	Remove Employee abstract class

5.3.6. Hypotheses Testing

To reiterate, we have formulated three hypotheses to address corresponding three research questions. Each hypothesis is focusing on a specific research question. The hypotheses are numerically validated as follows:

Hypothesis 4 (RQ1): In order to test this hypothesis, the Detection Recall (DR) of our detection approach is measured. The null hypothesis (H_{04}) can be rejected, if DR is significant. Numerically, it is set that if DR is greater than or equal to 80%, then null hypothesis (H_{04}) can be rejected. While executing on investigated sequence diagrams, our detection approach shows a significant DR of 90%. The DR is greater than 80%, so null hypothesis (H_{04}) is rejected. This answers our RQ1, that our proposed detection approach is able to detect significant number of the security bad smells in sequence diagrams.

Hypothesis 5 (RQ2): In order to test this hypothesis, the Correction Efficacy (CE) of our correction approach is measured. The null hypothesis (H_{05}) can be rejected, if CE is significant. In numerical terms, if CE is greater than or equal to 80%, then the null hypothesis (H_{05}) can be rejected. Our correction approach shows notable results by yielding a significant CE of 95% in investigated sequence diagrams. The CE is greater than 80%, so null hypothesis (H_{05}) is rejected. This addresses our RQ2 that our proposed correction approach is able to remove significant number of the security bad smells in sequence diagrams.

Hypothesis 6 (RQ3): The pair-wise t-test is performed to statistically analyze the significant security improvement in sequence diagrams. In our case, the pair-wise t-test is beneficial because it would be able to identify the differences in quality metrics as a result of refactoring. The p-value is computed with 95% confidence through a pair-wise t-test. The computed p-value is 0.04, which is less than 0.05. Hence, it can be concluded that security in investigated sequence diagrams has improved significantly. In succession to this observation, we can reject our formulated null hypothesis (H_{06}) with 95% confidence. This accomplishes the sub-goal of security improvement in sequence diagrams and answers our RQ3. For reference, the quality metrics values pre and post refactoring are provided in Appendix D.

5.4. Experimental Design for Class Diagrams

This section aims to provide details on our experiment with class diagrams. The following subsections describe our experiment in terms of experimental materials, variables, proposed hypotheses, experimental tasks, results and hypotheses testing.

5.4.1. Experimental Materials

Four open source java projects are used for validation of our detection approach. The projects include JGraphX, Cobertura, GanttProject, JHotDraw. The rationale for selecting these projects is their frequent consideration for bad smells investigation and refactoring [7, 8, 34, 99]. The projects are collected from online source [100]. Table 19 shows the statistics of the four selected projects in terms of a number of classes and number of each investigated security bad smell. JGraphX shows the relatively significant number of security bad smells in comparison with other projects. Overall, projects give a decent number of security bad smells instances for detection analysis. Similar to use case diagrams and sequence diagrams, other tools such as SDMetrics and Visual Studio are utilized. IntelliJIDEA and Eclipse are utilized for automated refactoring in class diagrams. In most of the refactoring, IntelliJIDEA is used because its effectiveness in handling bad smells.

Table 19. Statistics of analyzed projects

Projects	**Number of classes**	**Security bad smells**		
		Missing modularization	**Broken modularization**	**Deficient encapsulation**
JGraphX	294	20	5	8
Cobertura	132	3	4	2
GanttProject	866	5	3	5
JHotDraw	325	2	3	0

5.4.2. Variables

The dependent and independent variables are already described in section 5.2.2. The independent variable for detection of security bad smells in investigated class diagrams is detection recall. The measurement of how many security bad smells are removed as a result of refactoring is the independent variable for correction. The description of instances of security bad smells in investigated class diagrams is already provided in section 5.4.1. Quality metrics are useful in the quantitative validation of security improvement. The quality metrics selected for class diagram are as follows:

- *NAttr* is the number of attributes per class.
- *NOps* is the number of operations per class.
- *RPubAttr* is the ratio of public attributes to total attributes per class.
- *RPriAttr* is the ratio of private attributes to total attributes per class.
- *RProAttr* is the ratio of protected operations to total operations per class.
- *RPriOps* is the ratio of private operations to total operations per class.
- *RProOps* is the ratio of protected operations to total operations.
- *DIT* is the depth of inheritance.

- *NOC* is the number of children.
- *EC* is the number of instances in a class where the class is externally used.
- *IC* is the number of instances in a class where another class is referred to it.

Although many other quality metrics are computed to reach this set of metrics but only these are listed because they are directly used in definitions of the rules. For example, computation of RPubAttr requires a number of public attributes from total attributes, but it is not listed.

5.4.3. Proposed Hypotheses

Following hypotheses are formulated to statistically validate the effectiveness of our proposed approaches and make statistical judgment on security improvement in class diagrams:

Hypothesis 7 (RQ1): The proposed detection technique is able to identify a significant number of security bad smells in the investigated class diagrams.

Null Hypothesis (H_{07}): The detection approach is unable to identify a significant number of security bad smells in the investigated class diagrams as indicated by its recall.

Alternate Hypothesis (H_{17}): The detection approach is able to identify a significant number of security bad smells in the investigated class diagrams as indicated by its recall.

If the Detection Recall (DR) of detection technique in terms of identifying the security bad smells in investigated class diagrams is significant then the null hypothesis (H_{07}) can be rejected. For testing purpose, the quantification of our hypothesis is presented below in terms of detection recall:

Null Hypothesis (H_{07}): DR < 80%

***Alternate Hypothesis* (*H_{17}*):** DR >= 80%

Hypothesis 8 (RQ2): The correction technique is able to remove a significant number of security bad smells in the investigated class diagrams.

Null Hypothesis (H_{08}): The correction approach is unable to remove a significant number of security bad smells in the investigated class diagrams as indicated by its correction effectiveness.

Alternate Hypothesis (H_{18}): The correction approach is able to remove a significant number of security bad smells in the investigated class diagrams as indicated by its correction effectiveness.

If the Correction Efficacy (CE) of correction technique in terms of removing the security bad smells in investigated class diagrams is significant then the null hypothesis (H_{08}) can be rejected. For testing purpose, the quantification of our hypothesis is presented below in terms of correction efficacy:

Null Hypothesis (H_{08}): CE < 80%

***Alternate Hypothesis* (*H_{18}*):** CE >= 80%

Hypothesis 9 (RQ3): Refactoring to security bad smells improves the investigated class diagrams from a security perspective.

Null Hypothesis (H_{09}): No difference is observed in security quality of the investigated class diagrams as a result of refactoring to security bad smells as indicated by quality metrics.

Alternate Hypothesis (H_{19}): A Significant difference is observed in security quality of the investigated class diagrams as a result of refactoring to security bad smells as indicated by quality metrics.

The null hypothesis (H_{06}) is rejected in the case, where, quality metrics values before refactoring are not equal to quality metrics values after refactoring. For testing purpose, the quantification of our hypothesis is presented below in terms of p-value:

Null Hypothesis (H_{09}): p-value > 0.05

Alternate Hypothesis **(*H_{19}*):** p-value < 0.05

5.4.4. Experimental Tasks

Detection: Once again, the same process is applied for detection of security bad smells in class diagrams. The initial individuals are formed by governing rules from security bad smells existent in investigated class diagrams. The initial population is formed by aggregating the individuals. The population undergoes selection, crossover and mutation operations as described in section 4.2.2. Once genetic algorithm reaches its terminating condition, it yields a solution carrying best fitness.

Correction: The correction of security bad smells in class diagrams is achieved by using existing refactoring tools. A variety of class level refactoring tools is available, both open source and commercial. In our case, we preferred IntelliJIDEA tool because of its flexibility towards different refactoring techniques. Move method and extract class are mainly applied because of missing modularization and broken modularization security bad smells. These two refactoring techniques are found effective in eradicating missing modularization, while, broken modularization is corrected only through move method. This limits the delving of other classes in a given class. Deficient encapsulation is removed by changing access modifiers from public to private. This restricts outside access of data members of classes, ensuring security robustness.

Behavioral consistency: The behavioral consistencies, in the cases of broken modularization and missing modularization, are achieved in a similar manner to the sequence diagrams. In the case of refactoring to deficient encapsulation, the consistency is achieved by fulfilling the condition of accessing the data members of classes through setters and getters. Before refactoring, the data is available publically so can be accessed by any outside class. After refactoring, the access modifiers are changed to private, making the access restricted to the outside classes. The access after refactoring is enforced through the introduction of setters and getters methods. This way the behavior of the refactored class diagram remains consistent with the original class diagram.

5.4.5. Results

Detection: Once the GA finishes its execution with investigated class diagrams, it generates a set of rules, which represents the best solution. The best solution yielded by the genetic algorithm is shown in Figure 28. All three rules are measuring the bad smells by using IF-THEN conditional statements having quality metrics. R1, R2 and R3 are measuring for missing modularization, broken modularization and deficient encapsulation respectively. The gathering of quality metrics is performed automatically through SDMetrics tool except for the ones with ratios. The quality metrics comprising of ratios are calculated using MS Excel [101]. The rationales behind considered metrics and subsequent mapping of rules to specific smells are based on the descriptions provided by Fourati et.al [71] and Ouni et.al

[8]. If the metrics values of class(c) do not fulfill the threshold constraints set by these rules, then that class has corresponding security bad smell. The best solution (shown in Figure 28) is then applied on investigated java projects to evaluate its recall effectiveness. The set of rules governing best solution are able to identify 60 out of 60 security bad smells present in examined class diagrams, meaning, detection approach has 100% recall. This fulfills our RQ1, that our proposed detection approach is able to detect significant number of the security bad smells in class diagrams.

R1: IF (NOps(c) >= 21 AND (NOC(c) <= 1 OR DIT(c) <= 1) AND (EC(c) >= 3 OR IC(c) >=3)) THEN missing modularization(c)
R2: IF (NAttr(c) <= 7 AND NOps(c) <= 11 AND (DIT(c) == 0 OR NOC(c) == 0) AND EC(c) == 0) THEN broken modularization(c)
R3: IF (RPriAttr(c) <= 0.25 AND (RPubAttr(c) >= 0.88 OR RProAttr(c) >= 0.75) AND (RPriOps(c) == 0 OR RProOps(c) == 0)) THEN deficient encapsulation(c)

Figure 28. Best solution generated for class diagrams

Correction: The correction is validated manually as well as by running the existing detection tools on refactored class diagrams. The detection tools used for correction validation are InFusion and InCode. If a security bad smell remains in a project after appropriate refactoring application, it should be detected by the tools. In our experiment, no security bad smell emerged upon executing these tools on the refactored projects. This validates successful correction of security bad smells from investigated projects. Numerically, it means that 100% of the security bad smells are removed from four studied class diagrams.

5.4.6. Hypotheses Testing

To reiterate, we have formulated three hypotheses to address our three laid research questions. Each hypothesis is focusing on a specific research question. The hypotheses are numerically validated as follows:

Hypothesis 7 (RQ1): In order to test this hypothesis, the Detection Recall (DR) of our detection approach is measured. If the DR is significant, the null hypothesis (H_{07}) can be rejected. The threshold value set while formulating this hypothesis is 80%. This means that if the DR is greater than or equal to 80%, then null hypothesis (H_{07}) can be rejected. Our detection approach shows a significant DR of 100% after executing on investigated class diagrams. Since the DR is greater than 80%, so null hypothesis (H_{07}) is rejected. This fulfills our RQ1, that our proposed detection approach is able to detect significant number of the security bad smells in class diagrams.

Hypothesis 8 (RQ2): In order to test this hypothesis, the Correction Efficacy (CE) of our correction approach is measured. If CE is significant, then the null hypothesis (H_{08}) can be rejected. Our correction approach shows striking results by yielding a significant CE of 100% in investigated class diagrams. Since, the CE is greater than 80%, so null hypothesis (H_{08}) is rejected. This addresses our RQ2 that our proposed correction approach is able to remove significant number of the security bad smells in class diagrams.

Hypothesis 9 (RQ3): The pair-wise t-test is considered because the significant changes in metrics values (as a result of refactoring) are efficiently catered in it. The p-value will justify the significant security improvement in class diagrams. The p-value is computed with 95% confidence through a pair-wise t-test. The computed p-value is $6.44e10^{-11}$, which is less than 0.05. Hence, it can confidently be concluded that security in investigated class diagrams has improved significantly. As a result of this conclusion, we can reject our formulated null hypothesis (H_{09}) with 95% confidence. This fulfills another sub-goal of security improvement in class diagrams. For reference, the quality metrics values pre and post refactoring are provided in Appendix D. The metrics values are presented only for those classes having bad smells.

5.5. Summary of Hypotheses

At the end, all the hypotheses formulation and testing in our research are summarized in Table 20. It further eases the reader on comprehending the hypotheses collectively according to the laid research questions. It can be observed that Detection Recalls (DRs) of our proposed detection technique reach to their maximum limits in use case diagrams and class diagrams. This means that the solution generated by the detection approach is completely reliable in cases of these two diagrams. For sequence diagram, the detection technique also shows significant results and the reason for the drop of 10% in DR is discussed later in section 6.4.2. Almost similar results are observed for correction technique as well. The least p-value in class diagram is because of a number of components that are affected by security bad smells. Once the refactoring is applied, it leaves a major change in quality metrics values, making the p-value very small.

Table 20. Summary of hypotheses

Research questions	UML models	Hypotheses formulation	Hypotheses testing
RQ1	UC	H_{01}: DR < 80% H_{11}: DR > 80%	DR: 100%
	SD	H_{04}: DR < 80% H_{14}: DR > 80%	DR: 90%
	CD	H_{07}: DR < 80% H_{17}: DR > 80%	DR: 100%
RQ2	UC	H_{02}: CE < 80% H_{12}: CE > 80%	CE: 100%
	SD	H_{05}: DR < 80% H_{15}: DR > 80%	CE: 95%
	CD	H_{08}: DR < 80% H_{18}: DR > 80%	CE: 100%
RQ3	UC	H_{03}: p > 0.05 H_{13}: p < 0.05	p: 0.0001
	SD	H_{06}: p > 0.05 H_{16}: p < 0.05	p: 0.04
	CD	H_{09}: p > 0.05 H_{19}: p < 0.05	p: $6.44e10^{-11}$

UC: Use Case Diagram
SD: Sequence Diagram
CD: Class Diagram

5.6. Supplementary Experiments

The main purpose of supplementary experiments is to gain further confidence in the detection approach in terms of generated set of rules. Although, experiments for generation of detection rules are performed with three UML models and results are significant, but to further increase the confidence in our detection approach, we are performing experiments with large datasets. The abundance of security bad smells can strengthen the applicability of generated detection rules because the generation of detection rules rely on them. The supplementary experiments address this notion and justify on the generalization of detection rules. The supplementary experiments are performed with much bigger data size for three investigated UML models. This data is artificially generated because of unavailability of significant size data. The generation of data is performed in two ways: 1) simple replication 2) varied replication.

The large datasets should bring confidence in the generalization of rules and diversity of security bad smells. Once these datasets are input to the GA, they yield corresponding detection rules. We need to analyze whether the newly generated solutions are varied from previous best solution. Another concern to be noticed is the improvement in detection rules generated by our supplementary experiments. The improvement in detection rules is assessed by either enhanced detection recalls or refinement of rules in general. The newly generated rules (for both the replication cases) are applied on the same set of UML models as described in sections 5.2, 5.3 and 5.4.

5.6.1. Supplementary Experiments with Simple Replication Datasets

In simple replication, the data is produced by replicating the small datasets. For example, we initiate with existing four use case diagrams, then create the replications of these diagrams, making the dataset consisting of eight diagrams. The replication procedure halts when at least 1000 use case diagrams are created. The same process is used for replication in sequence diagrams and class diagrams. Table 21 shows the statistics of simply replicated datasets for investigated UML models. It can be seen that data sizes are significantly enhanced in terms of number of use cases/classes and security bad smells instances. It is made sure that at least 1000 use cases or classes are incorporated. Since the data sizes are increased, the security bad smells instances are automatically escalated. Dataset created through varied replication examines the flexibility of our GA as well.

Table 21. Statistics of simple replicated datasets

UML model	Number of use cases/classes	Security bad smells instances
Use case diagram	1160	638
Sequence diagram	1023	620
Class diagram	22638	840

Use case diagrams: The detection rules remain unchanged after performing the experiment with use case diagrams dataset. The rationale behind the stabilization of rules is the fewer number of components. Since the detection rules remain consistent, the Detection Recall (DR) remains same.

Sequence diagrams: In the case of sequence diagrams, the set of rules generated by our supplementary experiment with simple replication differs marginally from the best solution stated in section 5.3.5. Figure 29 states the best-generated solution as a result of our supplementary experiment. The differences are observed in R1, while R2 and R3 remain unmodified. The decrease is observed in NInvoc and NRec values. Due to the change in CBO value, the two missing modularization bad smells, undetected by previous best solution, are captured. The new best solution incorporates the lack of coupling in the rule and as a result the DR becomes 100%.

R1: IF (NAss(c) >= 11 AND (NInvoc(c) >= 4 AND NRec(c) >= 4) AND CBO(c) >= 1) THEN missing modularization(c)
R2: IF (NAss(c) == 2 AND (NInvoc(c) == 1 OR NRec(c) == 1) AND CBO(c) == 1) THEN broken modularization(c)
R3: IF (NAss(c) == 0 AND NInvoc(c) == 0 AND NRec(c) == 0 AND CBO(c) == 0) THEN unutilized abstraction(c)

Figure 29. Best solution for sequence diagrams (simple replication)

Class diagrams: Lastly, the supplementary experiments are executed with class diagrams having substantial data size. Since detection rules for class diagrams carry comparatively more number of quality metrics so more variations are expected. Once GA is applied, the produced best solution from simple replication is shown is Figure 30. Although, the best solution produced using small data set has significant DR of 100%, the new best solution seems to be more refined and likely to capture security bad smells more effectively. In R1, EC quality metric value is altered by a decrease of 1. The decrease in values would enforce the rule to capture more security bad smells. The new rule is also handling the external coupling more effectively. No change is observed in R2 because most of the metrics values are equal to zero, which is already the least value. In R3, there is a marginal decrease in RPubAttr metric and a marginal increase in RPriAttr. The increase in RPriAttr seems theoretically more promising as a greater number of deficient encapsulation instances can be identified. It is certain that new solution comprises of rules which would be able to detect more security bad smells. This is also supported by the achieved DR of 100%, when applied on class diagram data set summarized in section 5.4.1.

R1: IF (NOps(c) >= 21 AND (NOC(c) <= 1 OR DIT(c) <= 1) AND (EC(c) >= 2 OR IC(c) >=3)) THEN missing modularization(c)
R2: IF (NAttr(c) <= 7 AND NOps(c) <= 11 AND (DIT(c) == 0 OR NOC(c) == 0) AND EC(c) == 0) THEN broken modularization(c)
R3: IF (RPriAttr(c) <= 0.27 AND (RPubAttr(c) >= 0.86 OR RProAttr(c) >= 0.75) AND (RPriOps(c) == 0 OR RProOps(c) == 0)) THEN deficient encapsulation(c)

Figure 30. Best solution for class diagrams (simple replication)

5.6.2. Supplementary Experiments with Varied Replication Datasets

Varied replication opts for modifications in quality metrics values during replication. The reason of replicating with variations is to introduce more distinctive quality metrics values. For example, we initiate with existing four use case diagrams, then create the replications of these diagrams and modify the metrics values alongside. Now, the new dataset consists of eight different use case diagrams because of introduction of variations. The replication procedure halts when at least 1000 use case diagrams are created. Same process is used for replication in sequence diagrams and class diagrams. The statistics related to varied replication dataset are shown in Table 22. The datasets sizes are significantly increased in terms of number of use cases/classes. Since the data sizes are increased, the security bad smells instances are automatically escalated. Another reason of enhanced bad smells instances is the introduction of new instances because of variations. Dataset created through varied replication examines the flexibility of our GA as well.

Table 22. Statistics of replicated datasets with variations

UML model	Number of use cases/classes	Security bad smells instances
Use case diagram	1280	704
Sequence diagram	1056	640
Class diagram	25872	960

Use case diagrams: While performing the supplementary experiments with varied replication dataset for use case diagrams, the detection rules remain same. The experiments with the small dataset, simple replication dataset and varied replication dataset produce same detection rules. Once again, the rationale behind the stabilization of rules is the fewer number of components. Since the detection rules remain consistent, the Detection Recall (DR) remains same.

Sequence diagrams: In the case of the experiment having dataset produced from varied replication, two quality metrics values are modified in comparison with the best solution presented in section 5.3.5. Figure 31 states the best-generated solution as a result of our supplementary experiment with the varied dataset. The varied metrics are CBO and NInvoc of R1. The change in more quality metrics supports the variations in generated dataset. Similar to simple replication experiment, the best solution generated from varied replication carries DR of 100%. So we can conclude that the large datasets created from both types of replication have further refined the generated set of rules. This improvement in results contributes to strengthening the generalization of these rules.

R1: IF (NAss(c) >= 11 AND (NInvoc(c) >= 4 AND NRec(c) >= 5) AND CBO(c) >= 1) THEN missing modularization(c)
R2: IF (NAss(c) == 2 AND (NInvoc(c) == 0 OR NRec(c) == 1) AND CBO(c) == 1) THEN broken modularization(c)
R3: IF (NAss(c) == 0 AND NInvoc(c) == 0 AND NRec(c) == 0 AND CBO(c) == 0) THEN unutilized abstraction(c)

Figure 31. Best solution for sequence diagrams (varied replication)

Class diagrams: Since detection rules for class diagrams carry comparatively more number of quality metrics so more variations are expected because of dataset produced from varied replication. A comparatively greater number of quality metrics are altered in the experiment with dataset generated from varied replication. The basic rationale behind the greater alterations lies in the variations inserted during data replication. Once GA is applied, the produced best solution from the dataset (varied replication) is shown is Figure 32. In R1, three quality metrics (NOps, EC and IC) are modified. Both external and internal couplings are readjusted in the new rule. For R2, NAtrr and NOps are altered by a decrease. The decrease redefines the broken modularization in a more effective way because this bad smell appreciates the fewer number of attributes and operations. In R3, although three (RPriAttr, RPubAttr and RProAttr) quality metrics are fixated but a significant decrease in RPubAttr can be observed. The substantial decrease in RPubAttr is justifiable in a way that the previous value was too high and would ignore few legitimate security bad smells. Theoretically, it can be seen that new solution would be able to detect significant security bad smells. This is also supported by the acquired DR of 100%, when applied on class diagram data set summarized in section 5.4.1.

R1: IF (NOps(c) >= 20 AND (NOC(c) <= 1 OR DIT(c) <= 1) AND (EC(c) >= 2 OR IC(c) >=1)) THEN missing modularization(c)
R2: IF (NAttr(c) <= 5 AND NOps(c) <= 10 AND (DIT(c) == 0 OR NOC(c) == 0) AND EC(c) == 0) THEN broken modularization(c)
R3: IF (RPriAttr(c) <= 0.27 AND (RPubAttr(c) >= 0.79 OR RProAttr(c) >= 0.77) AND (RPriOps(c) == 0 OR RProOps(c) == 0)) THEN deficient encapsulation(c)

Figure 32. Best solution for class diagrams (varied replication)

As a conclusion, we can express that the supplementary experiments with large sets of UML data and security bad smells instances have improved few rules in generated solutions. The new sets of rules are also a refined form of previously acquired rules from small datasets in a way that they show a potential of capturing more security bad smells. This aids to more reliable and generalized set of rules.

5.7. Customization Guidelines for Proposed Detection Approach

This section provides the guidelines to assist the user of our automated detection approach. This section also aids the researchers on how they can benefit from the approach.

It would require two steps in order to detect security bad smells (considered in this research) on researchers' own UML models (use case diagram, sequence diagram and class diagram) using our generated rules. The first step includes the calculation of metrics values for their own models. The calculation of quality metrics can be accomplished as described in sections 5.3.1, 5.3.2 and 5.3.3 for use case diagrams, sequence diagrams and class diagrams respectively. Once the metrics values are available, in the second step, all it is required is to apply the rules already generated by our detection approach. The set of rules would detect security bad smells by comparing their metrics thresholds with the quality metrics values of the models under investigation. This process works if the researchers are interested in capturing the security bad smells investigated in our research.

Our detection approach is easily extensible if researchers are focusing on capturing other security bad smells. In order to achieve this objective, the detection approach needs to be re-executed. Different security bad smells would require a different set of quality metrics as well. This time, the detection approach forms the initial set of rules for a different set of security bad smells using a different set of quality metrics. The set of rules will go through selection, crossover and mutation operations in a similar manner as described in section 4. Once GA finishes its execution, it will yield a set of rules as the best solution. The generated set of rules can subsequently be applied to the investigated UML models to detect the considered security bad smells.

CHAPTER 6

ANALYSIS AND DISCUSSION

This chapter focuses on the implications of our research. The analysis and discussion justify many expected propositions. The justifications for the use of security bad smell examples and their abundance in investigated UML models are presented in this chapter. The discussion on consideration of quality metrics and their values is provided as well. This chapter also analyzes the impact of refactoring to security on other quality attributes.

6.1. Security Bad Smells

In our detection approach, the most important ingredient is security bad smells examples because the formulation of rules mainly depends on it. The quality of the solution is contingent on the quality of base examples. During individual formulation, the diversity is ensured by selecting rules wisely. This is depicted during detection validation that the yielded solution is able to detect a significant number of security bad smells. In addition, an abundance of investigated security bad smells has allowed us to draw solutions with the maximum recall.

Bad smells examples are in abundance in online software repositories. Sometimes bad smells are reported in maintenance directory, if not, they can be identified manually or using existing tools. The bad smells examples allow catering the actual programming practices in the detection process. As a result, the yielded rules become more precise and context faithful. The examples also remove the existing contradictions about metrics threshold values as it solves the subtleness of agreeing on a commonly accepted metrics values. The rules generation process is executed multiple times using bad smells examples to erase any uncertainty with respect to the quality of rules.

It can be argued that work overhead exists using security bad smells examples because they need to be identified before the start of the GA. The rationale for using base examples is to remove any confusions of quality metrics thresholds. It would have been a major threat to validity if thresholds would have been used instead of base examples. The consensus upon quality metrics thresholds would have taken this study to unjustifiable arguments. The use of base examples is completely justified in our study. Another reason for using base examples is to incorporate real programming mistakes that lead to security bad smells. Another consideration is the dependency of our detection approach on the size of base examples set. Our experiment answers this argument by showing significant results using a small set of base examples.

Our study is not biased towards a specific security bad smell. It is made sure that each investigated case study of UML models carries a balanced number of security bad smells. Only one instance opposes this claim and that is missing modularization bad smell in JGraphX. This is because of an actual larger number of missing modularization defect in JGraphX. During rules formulation, it is guaranteed that all investigated diagrams are given relatively equal weight.

Although it is assured that decent frequency of security bad smells exists in investigated UML models, however, the number of instances varies among different bad smells. For example, in use case diagrams, missing hierarchy is more abundant than other bad smells. The reason is the inappropriate associations between actors and use cases. The security bad smells found in studied sequence diagrams are evenly distributed. In other words, the frequencies of different types of bad smells are almost equal. In the case of class diagrams, missing modularization is found plentiful. The rationale behind is the common practice of exercising less modularization at design and implementation levels. It is mostly practiced that classes are overburdened and relevant concerns are not separated.

6.2. Consistency of Results

Another concern to discuss here is the consistency of our results. The acquired results are consistent because our detection approach caters quantitative information using quality metrics. If the semantics of investigated UML models are also considered, then consistency would have been an issue to address. In our case, the approach is irrelevant to UML models semantics so consistency of results is automatically achieved.

In order to improve the consistency of results, we have performed experiments with large datasets of three investigated UML models. Although experiments with small datasets give significant results, experiments with large datasets further improve the results in terms of consistency. The consistency can be observed from the achieved detection recall while experimenting with large datasets. The experiments also help in the refinement of the detection rules. The same is confirmed by the detection recall of our detection approach as well. The experiments with varied replications produce comparatively better detection rules. The rationale behind is the diversity in the datasets because of the modifications during replications.

The impact of different programming languages and paradigms might affect the results of our study. Different programming practices and evolution of designs might mildly change the solution generated by our detection approach. We claim that our approaches are stable. The generated solution from multiple runs of GA almost yielded solutions with none or minimal difference in fitness. During correction, the XMI transformations from UML models remain consistent.

6.3. Variations in Quality Metrics

During validation of security improvement, the changes in quality metrics values are observed. Though all quality metrics contribute towards the security improvement in a specific UML model but the impact of metrics may vary depending on the security bad smell being removed. For instance, the refactoring to missing modularization would bring significant changes to the metrics values because the class undergoes decomposition. On the other hand, refactoring broken modularization marginally changes the metrics. Another important point to discuss is the trend of variation in metrics. It is observed that, while refactoring, the metrics values tend to decrease. This means that lower values of metrics are desirable to have a more secure UML model. The investigated security bad smells are also of nature, when refactored, lean to decrease the metrics values.

6.4. Impact of Applied Refactoring on Quality Attributes

Although we have analyzed the effect of refactoring on security improvement, the UML models have shown improvements in other quality properties as well. The notable enhancement in quality is observed in terms of modularity, complexity, reusability and design size.

6.4.1. Impact in Use Case Diagrams

The refactoring to broken modularization in use case diagram reduces the number of inclusion use cases, which suggests the model is less complex. The refactoring of inclusion use cases has also reduced the quantity of use cases per actor, which means that the model size is decreased. The refactoring to missing modularization bad smell introduced modularity in the use case diagrams. This acknowledges the separation of concerns property as well. The increased modularity in use case diagrams allowed reduced complexity, enhanced reusability.

6.4.2. Impact in Sequence Diagrams

The quality improvements in sequence diagrams are also observed in a similar manner. The issue of unutilized abstraction is resolved through the removal of the abstraction. This not only decreases the design size but also ensures the correct operational behavior. The problem of broken modularization is solved by movement of method and removal of the respective class. The movement of method strengthens the modularization; and removal of class contributes to the reduction of design size. The overall number of messages are also reduced. The major reduction in design size comes from the removal of lifelines as result of refactoring to unutilized abstraction and broken modularization. The refactoring to missing modularization reduced the number of messages between two classes. The burden of interactions between two classes is shared by a newly introduced class. This way the model is modularized, which means less complexity and more reusability. The separation of concerns is also validated since classes now only deal with what concern them. The two undetected missing modularization bad smells in sequence diagrams are because of lack of coupling. It is normally perceived that if a component possesses missing modularization, it exhibits high coupling. But in two cases, the coupling was low regardless of missing modularization problem. The low coupling restricted the rule from detecting the bad smells. The generated rules with large datasets handle low coupling and those rules are able to detect two undetected bad smells in sequence diagrams as well.

6.4.3. Impact in Class Diagrams

Similarly, the quality improvement happened in class diagrams. The three investigated security bad smells are refactored to improve security, and in connection, other quality attributes, such as modularity, complexity, design size and reusability, are refined as well. Broken modularization results in movement of method or methods to the appropriate class. The modularization is reshaped to achieve better application of modularization concept in object oriented design. Better modularization also encourages reusability of a design. Missing modularization violates the imperative object oriented properties such as separation of concerns, modularity and reusability. Upon refactoring missing modularization smells, the design becomes more flexible and carries these important properties. Eradication of missing and broken modularizations reduce the degree of coupling as well. Deficient encapsulation is a unique security bad smell as it concerns with only a class itself, meaning the effect of refactoring is confined to the class being refactored. In other words, the refactoring to this bad smell does not affect other classes. The uniqueness of deficient encapsulation can also be observed from its weary impact on quality attributes like modularity, complexity, design size and reusability.

It can be extracted that as a byproduct of the refactoring to security bad smells, many other quality attributes are improved. The quality upgrade is observed in all investigated UML models. Modularity, complexity, design size and reusability are the quality attributes that showed quality revamp. In addition, the introduction of these quality attributes eases the analysis of UML models as well.

CHAPTER 7

THREATS TO VALIDITY

This chapter reports the validity threats and how they are dealt to minify their impact on experimental validation of our proposed techniques. The most common classification to address validity threats is construct validity, conclusion validity, internal validity and external validity and is adopted to report the validity threats of our research.

7.1. Construct Validity

The most important activity in the experimental process is the correct selection of independent variables. It needs to be closely judged that selected independent variables are correlated with the dependent variable. In our experimental validation, the independent variables i.e. quality metrics are selected based on previous studies and after in-depth analysis to ensure their effectiveness in measuring the security aspects in UML models. Even though the mapping of quality metrics is done according to the published literature, but there still exists feeble possibility that we might have overlooked an imperative metric for specific bad smell measurement. Another construct validity threat is connected with security bad smells examples. Some main security bad smells examples might be overlooked during individual formulation. This threat is reduced by selecting the security bad smells examples with extreme concentration. It also does not affect the results as much because of crossover and mutation operations. The suspicions about biasness of experimental outcomes are totally removed by laying no pre-expectations from our experiments.

7.2. Conclusion Validity

The conclusions drawn as a result of our experiments are based on sufficient subjective and objective findings. The objectivity of quality metrics has allowed us to reach meaningful and definite conclusions. The supreme objectivity of quality metrics has encouraged us to incorporate them in our empirical validation and as expected, they have contributed significantly to our conclusions. The automated collection of quality metrics for use case diagrams and class diagrams significantly enhances the accuracy of the computed values and minifies the conclusion validity threat. For sequence diagrams, the quality metrics are manually calculated with absolute care, but there is always a threat posed by manual computation. This threat is minified by computing the quality metrics multiple times. The replications of datasets are also performed manually so it may cause certain threats to the conclusion validity. This threat is minimized by making the replication randomly. The randomization has introduced diversity in the datasets, which is the ultimate objective, regardless of manual replication.

7.3. Internal validity

The analyzed UML models are not exposed to any treatment except correction to observe only the influence of refactoring on them. No modifications in treatments are made to observe findings under similar conditions. The post refactoring states of UML models are carefully saved for the computation of quality metrics. The import and export of UML models to and from XML are performed using the same tool to avoid any structural change in XML representations. The modifications in XML representations are performed manually but it does not really impact the validity because the corresponding exported UML models are validated with the expected refactored UML models.

7.4. External validity

The external validity usually poses threats to the generalization of results. To mitigate this threat, a favorable number of models case studies are collected and are a good representation of actual UML models. The generalization of results is also improved because validation is also performed with large datasets carrying a significant number of security bad smells instances. The investigated class diagrams are extracted from Java projects so the applicability of the results to other object-oriented languages might be constrained. Some language-dependent aspects may vary while extracting corresponding class diagrams.

CHAPTER 8

CONCLUSION AND FUTURE WORK

A number of quality attributes related to software design have been reported in the literature, such as modularity, reusability, modifiability, testability, security etc. The majority of the quality attributes have been studied rigorously to assess the impact of refactoring on the improvement of software quality. However, there is a scarcity of corpus on investigating the contribution of refactoring in improving security aspects of software models. It is imperative to analyze software models from a security point of view as well.

In our research, we overcome the problem of security in UML models (class diagram, sequence diagram and use case diagram) by the application of refactoring. The detection of security bad smells is achieved through the adaptation of a genetic algorithm, while, correction is accomplished by model transformation approach. The detection approach uses security bad smells instances and quality metrics to formulate rules. The best set of rules generated by GA is used for detection of security bad smells in studied UML models. The correction approach applies a model transformation using XMI for refactoring of identified security bad smells in investigated UML models.

The proposed approaches are validated by performing experiments with multiple case studies of investigated UML models. Our detection approach is able to detect security bad smells with 100% recall in investigated use case diagrams and class diagrams, and 90% recall in investigated sequence diagrams. The correction approach also shows extraordinary results by removing 100% security bad smells by refactoring application in investigated use case diagrams and class diagram, and 95% in investigated sequence diagrams. We also performed supplementary experiments with large datasets to generate more generalized detection rules because our detection approach relies heavily on generated rules. The sets of rules generated by supplementary experiments are improved in terms of detecting more legitimate security bad smells. Through statistical analysis of quality metrics, we are also able to conclude on the significant improvement in security quality of investigated UML models as result of refactoring.

The compelling results delivered by our detection and correction approaches have encouraged us to extend our work in future. We plan to apply our approaches to other UML models to gain further confidence on their applicability to other models. We also plan to apply same approaches with a different set of security bad smells to assess their appropriateness with other security bad smells. We intend to evaluate our approaches with projects developed using other programming languages to enhance the generalization of acquired results for UML class diagrams.

Appendix A: Code and Model Refactoring

Sr. No.	Bad Smell	Code	Design	Both
1	Add parameter			Yes
2	Change bidirectional association to unidirectional			Yes
3	Change reference to value	Yes		
4	Change unidirectional association to bidirectional			Yes
5	Change value to reference	Yes		
6	Collapse Hierarchy			Yes
7	Consolidate conditional expression	Yes		
8	Consolidate duplicate conditional fragments	Yes		
9	Decompose conditional	Yes		
10	Duplicate observed data			Yes
11	Dynamic method definition		Yes	
12	Eagerly initialized attribute	Yes		
13	Encapsulate collection		Yes	
14	Encapsulate downcast	Yes		
15	Encapsulate Field	Yes		
16	Extract Class			Yes
17	Extract interface			Yes
18	Extract Method			Yes
19	Extract Module	Yes		
20	Extract subclass			Yes
21	Extract superclass			Yes
22	Extract surrounding method.	Yes		
23	Extract variable	Yes		
24	Form template method			Yes
25	Hide delegate			Yes
26	Hide method			Yes
27	Inline class			Yes
28	Inline method	Yes		
29	Inline module	Yes		
30	Inline temp	Yes		
31	Introduce assertion	Yes		
32	Introduce class annotation	Yes		
33	Introduce expression builder			Yes
34	Introduce foreign method			Yes
35	Introduce gateway			Yes
36	Introduce local extension			Yes
37	Introduce named parameter	Yes		
38	Introduce null object			Yes
39	Introduce Parameter object			Yes
40	Isolate dynamic receptor			Yes
41	Lazily initialized attribute	Yes		
42	Move eval from runtime to parse time	Yes		
43	Move field			Yes
44	Move method			Yes
45	Parameterize method			Yes
46	Preserve whole object		Yes	
47	Pull up constructor body	Yes		
48	Pull up field			Yes
49	Pull up method			Yes
50	Push down field			Yes

51	Push down method			Yes
52	Recompose conditional	Yes		
53	Remove assignment to parameters	Yes		
54	Remove control flag	Yes		
55	Remove middle man			Yes
56	Remove named parameter	Yes		
57	Remove parameter			Yes
58	Remove setting method			Yes
59	Remove unused default parameter	Yes		
60	Rename method			Yes
61	Replace abstract superclass with module			Yes
62	Replace array with object	Yes		
63	Replace conditional with polymorphism	Yes		
64	Replace constructor with factory method	Yes		
65	Replace data value with object			Yes
66	Replace delegation with hierarchy			Yes
67	Replace delegation with inheritance			Yes
68	Replace dynamic receptor with dynamic method	Yes		
69	Replace error code with exception	Yes		
70	Replace exception with test	Yes		
71	Replace hash with object	Yes		
72	Replace inheritance with delegation			Yes
73	Replace loop with correction closure method	Yes		
74	Replace magic number with symbolic constant	Yes		
75	Replace method with method object	Yes		
76	Replace nested conditional with guard clauses	Yes		
77	Replace parameter with explicit method	Yes		
78	Replace parameter with method	Yes		
79	Replace record with data class			Yes
80	Replace subclass with fields			Yes
81	Replace temp with chain	Yes		
82	Replace temp with query	Yes		
83	Replace type code with class			Yes
84	Replace type code with module extension	Yes		
85	Replace type code with polymorphism			Yes
86	Replace type code with state/strategy			Yes
87	Replace type code with subclasses			Yes
88	Self-encapsulate field	Yes		
89	Separate query from modifier			Yes
90	Split temporary variable	Yes		
91	Substitute algorithm	Yes		
92	Duplicated Code	Yes		
93	Long Method	Yes		
94	Large Class			Yes
95	Long Parameter List			Yes
96	Divergent Change	Yes		
97	Shotgun Surgery			Yes
98	Feature Envy			Yes
99	Data Clumps			Yes
100	Primitive Obsession	Yes		
101	Switch Statements	Yes		
102	Parallel Inheritance hierarchies			Yes
103	Lazy Class			Yes

104	Speculative Generality	Yes		
105	Temporary Field	Yes		
106	Message Chain			Yes
107	Middle Man			Yes
108	Inappropriate Intimacy			Yes
108	Alternative classes with Different Interfaces	Yes		
110	Incomplete Library Class	Yes		
111	Data class			Yes
112	Refused Bequest			Yes
113	Comments	Yes		
114	God class or the Blob			Yes
115	Functional Decomposition			Yes
116	Poltergeist			Yes
117	Swiss army knife			Yes
118	Lava flow	Yes		
119	Spaghetti code	Yes		
120	Type Checking	Yes		
121	Poor use of abstraction		Yes	
122	Hidden concurrency		Yes	
123	Unnecessary behavioral complexity		Yes	
124	Too low cohesion		Yes	
125	Too strong coupling		Yes	
126	Structural complexity		Yes	
127	Specialization aggregation		Yes	
128	Missing Abstraction			Yes
129	Imperative Abstraction			Yes
130	Incomplete Abstraction			Yes
131	Unnecessary Abstraction			Yes
132	Unutilized Abstraction.			Yes
133	Duplicate Abstraction			Yes
134	Deficient Encapsulation			Yes
135	Leaky Encapsulation			Yes
136	Missing Encapsulation			Yes
137	Unexploited Encapsulation.	Yes		
138	Broken Modularization			Yes
139	Insufficient Modularization			Yes
140	Cyclically dependent Modularization			Yes
141	Hub-like Modularization			Yes
142	Missing Hierarchy			
143	Unnecessary Hierarchy		Yes	
144	Unfactored Hierarchy			Yes
145	Wide Hierarchy			Yes
146	Speculative Hierarchy			Yes
147	Deep Hierarchy			Yes
148	Rebellious Hierarchy			Yes
149	Broken Hierarchy			Yes
150	Multipath Hierarchy			Yes
151	Cyclic Hierarchy			Yes

Appendix B: Opdyke's Identified Refactoring

Sr. No	Low-Level Refactoring	High-Level Refactoring
1	Create empty class	Create common superclass
2	Create member variable	Function signature compatible
3	Create member function	Adding function signatures to superclass
4	Delete unreferenced class	Makin function body compatible
5	Delete unreferenced variable	Making variables compatible
6	Delete unreferenced functions	Migrate variable to superclass
7	Change class name	Migrate common code to superclass
8	Change variable name	Identify class in-variants
9	Change membership function name	Create subclass and assign class in-variants
10	Change type	Simplify conditional statements and migrate to subclass
11	Change access control mode	Specialize expressions that create instances
12	Add function argument	Check a member as a component
13	Reorder function arguments	Add member to a set of component variable
14	Add function body	Remove members from set of component variables
15	Delete function body	Moving members into component
16	Convert instance variable to pointer	Moving members into aggregate
17	Convert variable reference to function Call	
18	Replace Statement list with function Call	
19	Incline function call	
20	Change superclass	
21	Move member variable to superclass	
22	Move member variable to subclass	
23	Abstract Access to member variable	
24	Convert code segment to function	
25	Move class	

Appendix C: Taxonomy of Security Bad Smells [3, 29, 54, 55]

Bad Smell	Description	Security Violation	Refactoring
Missing Abstraction	In the absence of an abstraction, the data and behavior are spread across design.	Confidentiality, Secrecy, Guarded Access, Integrity, Insecure Info Flow.	Replace type-code with class.
Incomplete Abstraction	When an abstraction does not support complementary or interrelated methods.	Correctness, Integrity	Introduce the missing complementary operation(s)
Multifaceted Abstraction	When abstraction is assigned more than one responsibility.	Correctness, Integrity	Extract class
Unutilized Abstraction	When an unused abstraction is accidentally invoked, it may result in runtime problems.	Integrity, Reliability	Remove unutilized abstraction.
Duplicate Abstraction	When two abstractions have same names, it is confusing which abstraction to invoke.	Non-repudiation, Integrity.	Rename abstraction, remove abstraction.
Deficient Encapsulation	It provides direct access of class data to outside classes.	Confidentiality, Secrecy, Guarded Access, Integrity.	Encapsulate field
Leaky Encapsulation	When internal data structures are leaked, the integrity of abstraction may be compromised	Confidentiality, Secrecy, Guarded Access, Integrity.	Encapsulate field and methods (if necessary)
Missing Encapsulation	when implementation variations are not encapsulated	Confidentiality, Secrecy, Guarded Access, Integrity.	Encapsulate field and methods (if necessary)
Broken Modularization	The data and related procedures are split across abstractions	Confidentiality, Secrecy, Guarded Access, Integrity.	Move method/Field
Cyclically dependent Modularization	Changes to a cyclically dependent abstraction can lead to runtime problems across other abstractions	Integrity	Move method or field
Missing Modularization	when a class is not decomposed	Reliability, Correctness	Extract class, Move methods
Rebellious Hierarchy	when a subtype rejects the methods provided by its super-type	Reliability, Correctness, Integrity.	Apply move method from the super-type to the relevant subtypes
Unnecessary Hierarchy	When inheritance is applied unnecessarily for a particular design context	Integrity, Insecure Info Flow.	Collapse hierarchy
Missing Hierarchy	To explicitly manage variation in hierarchical behavior, where a hierarchy could have been created and used to encapsulate those variations	Reliability, Integrity	Connection with appropriate hierarchy interface should be made
Broken Hierarchy	When developers are not aware that the super-type and subtype do not share an IS-A relationship	Confidentiality, Secrecy, Guarded Access, Insecure Info Flow, Integrity.	Replace inheritance with del-egation

Appendix D: Quality Metrics Values Pre and Post Refactoring

Quality Metrics Values for Use Case Diagrams

System	Use case	Before refactoring					After refactoring				
		NumAss	Including	Included	Extended	Extending	NumAss	Including	Included	Extended	Extending
Travel Agency	Arrange Tour	2	0	0	1	0	2	0	0	1	0
	Book Airline Tickets	3	0	0	0	2	0	0	0	0	1
	Book Airline Tickets 2						0	0	0	0	1
	Book Tour	1	0	0	1	0	1	0	0	1	0
	Deliver Airline Tickets	1	0	0	0	0	1	0	0	0	0
	Pay Commission	0	0	0	0	3	0	0	0	0	1
	Pay Commission 2						0	0	0	0	1
	Pay Commission 3						0	0	0	0	1
	Pay for Ariline Tickets	1	0	1	1	1	0	0	0	1	1
	Pay Travel Agent	2	1	0	1	0	2	0	0	1	0
	Pay for Tour	1	0	0	2	0	1	0	0	2	0
	Reserve Seat	1	0	0	0	0	1	0	0	0	0
HR System	Elect Reimbursement	1	0	0	0	1	0	0	0	0	1
	Elect Stock Purchase	1	0	0	0	3	0	0	0	0	1
	Elect Stock Purchase 2						0	0	0	0	1
	Elect Stock Purchase 3						0	0	0	0	1
	Issue Purchase Invoice	1	0	0	1	0	1	0	0	1	0
	Provide Stock	1	0	0	1	0	1	0	0	1	0
	Update Benefits	1	3	0	2	0	1	0	0	2	0
	Update Dental Plan	0	0	1	0	0					
	Update Insurance Plan	0	0	1	0	0					
	Update Medical Plan	0	0	1	0	0					
Restaurant System	Chinese	0	0	0	0	2	0	0	0	0	1
	Chinese 2						0	0	0	0	1
	Cook Food	1	0	0	0	0	1	0	0	0	0
	Drink Wine	1	0	0	0	1	0	0	0	0	1
	Eat Food	1	0	0	1	0	1	0	0	1	0
	Indian	0	0	0	0	2	0	0	0	0	1
	Indian 2						0	0	0	0	1
	Italian	0	0	0	0	2	0	0	0	0	1
	Italian 2						0	0	0	0	1
	Order Food	3	0	0	4	0	3	0	0	4	0
	Order Wine	3	0	0	0	1	0	0	0	0	1
	Pay for Wine	2	0	0	0	1	0	0	0	0	1
	Pay Tip	0	0	1	0	0					
	Pay for Food	3	1	0	4	0	3	0	0	4	0
	Serve Food	1	0	0	1	0	1	0	0	1	0
	Serve Wine	1	0	0	0	1	0	0	0	0	1
ATM	Bad Pin	0	0	0	0	2	0	0	0	0	1
	Bad Pin 2						0	0	0	0	1
	Balance Check	0	0	0	0	0	0	0	0	0	0
	Deposit	0	0	0	0	0	0	0	0	0	0
	Login	1	0	2	0	0	1	0	2	0	0
	Print Receipt	0	0	0	0	0	0	0	0	0	0
	System Maintenance	1	1	0	2	0	1	2	0	2	0
	System Reporting	1	0	0	0	1	0	0	0	0	1
	System Shutdown	0	0	1	0	0					
	Transaction	1	1	0	1	0	1	1	0	1	0
	Withdraw	0	0	0	0	0	0	0	0	0	0

Quality Metrics Values for Sequence Diagrams

System	Class	Before refactoring				After refactoring			
		NAss	NInvo	NRec	CBO	NAss	NInvo	NRec	CBO
Airline Reservation	Customer	15	7	8	1	9	4	5	1
	Booking System	23	5	3	3	21	5	2	3
	Flight	2	0	1	1				
	Customer DB	6	0	4	1	6	0	4	1
	Reservation System	0	0	0	0				
	Visitor					6	3	3	1
Movie Ticketing	Admin	6	3	3	1	6	3	3	1
	Registered User	11	5	6	4	8	3	4	3
	Visitor	0	0	0	0				
	Movies	8	0	4	2	7	1	3	2
	Book Ticket	4	0	2	1	4	0	2	1
	Payment	3	1	1	1	3	1	1	1
	Cancel Ticket	2	0	1	1				
	Ticket	0	0	0	0				
School Management	Admin	11	6	5	3	6	3	3	1
	Student	12	1	6	2	12	1	6	2
	Teacher	9	3	5	2	9	3	5	2
	Employee	0	0	0	0				
	Class	2	0	1	1				
	Department					3	2	1	1
Library Management	Librarian	38	11	19	3	26	6	13	2
	Books	10	4	5	1	10	4	5	1
	User	20	4	10	1	16	3	8	1
	Publisher	10	1	5	2	8	0	4	1
	Staff	0	0	0	0				
	Manager	2	0	1	1				
	Assistant					12	5	6	2
	Premiun User					4	1	2	1
Hotel Management	Manager	4	1	2	2	2	0	1	1
	Stock	2	0	1	1				
	Receptionist	17	5	9	3	9	3	5	2
	Customer	13	5	6	3	7	2	3	2
	Chef	4	2	2	3	1	0	1	1
	Food Items	2	0	1	1				
	Room Attendent	2	1	1	1				
	Room	4	0	2	1	4	0	2	1
	Staff	0	0	0	0				
	Assistant					6	1	3	3
	Resident					5	3	2	2

Quality Metrics Values for Class Diagrams Pre-refactoring

Class	Before refactoring																
	NumAttr	NumOps	numPubAttr	NumPriAttr	numProAttr	NumPubOps	numPriops	numProOps	ratioPubAttr	ratioPriAttr	ratioProAttr	ratioPubOps	ratioPriOps	ratioProOps	NOC	DIT	EC_Par	IC_Par
mxGraphView	7	61	0	1	6	61	0	0	0.14	0.86	0	1	0	0	0	1	10	77
mxGraphHandler	32	53	3	1	28	41	0	12	0.09	0.03	0.88	0.77	0	0.23	0	1	2	8
mxUtils	3	87	2	0	1	87	0	0	0.67	0	0.33	1	0	0	0	0	0	34
mxEdgeHandler	6	30	0	0	6	14	0	16	0	0	1	0.5	0	0.5	1	1	0	7
mxCellStatePreview	6	21	0	0	6	16	0	5	0	0	1	0.76	0	0.24	0	0	3	18
mxGraphics2DCanvas	7	22	3	0	4	20	0	2	0.43	0	0.57	0.9	0	0.1	1	1	65	8
mxCoordinateAssignment	28	30	0	1	27	14	0	16	0	0.04	0.96	0.47	0	0.53	0	0	0	20
mxCellEditor	25	23	3	3	19	19	0	4	0.12	0.12	0.76	0.83	0	0.17	0	0	0	5
mxMovePreview	13	24	0	0	13	19	0	5	0	0	1	0.79	0	0.21	0	1	2	8
mxGraphTransferHandler	11	22	2	1	8	17	0	5	0.18	0.09	0.72	0.77	0	0.23	0	0	1	13
mxConnectionHandler	19	40	1	1	17	37	0	3	0.05	0.05	0.9	0.93	0	0.07	0	1	2	12
mxGraphHierarchyModel	8	22	4	1	3	12	0	1	0.5	0.125	0.375	0.92	0	0.08	0	0	15	11
mxGraphOutline	24	25	1	1	22	22	0	3	0.04	0.04	0.92	0.88	0	0.12	0	0	1	3
mxSelectionCellsHandler	10	22	1	1	8	20	0	2	0.1	0.1	0.8	0.9	0	0.1	0	0	2	5
mxCellMarker	14	38	2	1	11	32	0	6	0.14	0.07	0.78	0.84	0	0.16	1	0	5	19
mxGraphStructure	3	27	0	3	0	26	0	1	0	1	0	0.94	0	0.04	0	0	0	27
mxGraphModel	8	85	0	0	8	70	0	15	0	0	1	0.82	0	0.18	0	1	0	36
mxCellHandler	8	32	0	0	8	22	0	10	0	0	1	0.69	0	0.31	2	0	2	5
BasicGraphEditor	13	30	0	1	12	19	0	11	0	0.07	0.93	0.63	0	0.37	2	0	16	7
mxFastOrganicLayout	22	24	0	0	22	20	0	4	0	0	1	0.83	0	0.17	0	1	0	3
ClassMap	12	22	0	12	0	20	0	2	0	1	0	0.9	0	0.1	0	0	6	2
InstrumentTask	14	21	0	14	0	16	0	5	0	1	0	0.76	0	0.24	0	1	0	6
HTMLReport	6	22	0	6	0	1	21	0	0	1	0	0.05	0.95	0	0	0	0	13
GanttProject	30	69	0	23	7	63	0	6	0	0.76	0.24	0.91	0	0.09	0	1	20	29
TaskManagerImpl	20	66	0	20	0	44	16	4	0	1	0	0.67	0.24	0.06	0	0	7	57
ChartModelBase	28	53	1	23	4	39	6	8	0.04	0.82	0.14	0.74	0.11	0.15	2	0	11	28
GanttOptions	35	76	3	32	0	68	8	0	0.09	0.91	0	0.89	0.11	0	0	1	3	19
UIFacadeImpl	22	55	0	22	0	41	11	3	0	1	0	0.75	0.2	0.05	0	0	2	25
XMLElement	15	71	2	13	0	47	0	24	0.13	0.87	0	0.66	0	0.34	0	0	1	11
DefaultDrawingView	17	66	0	17	0	52	5	9	0	1	0	0.79	0.07	0.13	0	0	0	23
EditorMenuBar	1	3	0	1	0	3	0	0	0	1	0	1	0	0	0	0	0	3
mxVmlCanvas	1	10	0	0	1	10	0	0	0	0	1	1	0	0	0	1	0	2
mxHtmlCanvas	1	10	0	0	1	10	0	1	0	0	1	0.9	0	0.1	0	1	0	2
SetLabelPositionAction	2	2	0	0	2	2	0	0	0	0	1	1	0	0	0	0	0	0
mxChildChangeCodec	0	6	0	0	0	6	0	0	0	0	0	1	0	0	0	1	0	8
CodeInstrumentationTask	4	8	0	4	0	1	7	0	0	1	0	0.13	0.87	0	0	0	0	7
Cobertura	5	9	0	7	0	7	2	0	0	1	0	0.78	0.22	0	0	0	0	3
ReportMain	1	6	0	1	0	2	4	0	0	1	0	0.33	0.67	0	0	0	0	0
HTMLReportFormatStrategy	1	2	0	1	0	2	0	0	0	1	0	1	0	0	0	0	0	1
OverwritingMerger	2	6	0	2	0	3	3	0	0	1	0	0.5	0.5	0	0	0	0	8
DayGridSceneBuilder	4	10	0	4	0	2	8	0	0	1	0	0.2	0.8	0	0	1	0	5
CalendarSaver	2	2	0	2	0	0	2	0	0	1	0	0	1	0	0	1	0	1
VerticalLayouter	0	2	0	0	0	2	0	0	0	0	0	1	0	0	0	1	0	2
HorizontalLayouter	0	2	0	0	0	2	0	0	0	0	0	1	0	0	0	1	0	2
NodeFigure	5	10	0	5	0	8	1	1	0	1	0	0.8	0.1	0.1	0	3	0	5
mxConstants	188	0	188	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
mxSwingConstants	17	0	17	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
mxGeometry	8	19	1	1	6	19	0	0	0.12	0.12	0.77	1	0	0	0	2	23	11
mxPngSuggestedPaletteEntry	8	0	7	1	0	0	0	0	0.88	0.12	0	0	0	0	0	0	1	0
ParseException	3	7	0	0	3	7	0	0	0	0	1	1	0	0	0	0	2	0
mxImage	4	7	0	1	3	7	0	0	0	0.25	0.77	1	0	0	0	0	0	0
mxEvent	57	0	57	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
mxEdgeStyle	29	0	29	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
AbstractCoberturaTestCase	6	7	6	0	0	7	0	0	1	0	0	1	0	0	12	0	0	0
ObjectMetric	3	2	3	0	0	2	0	0	1	0	0	1	0	0	0	1	0	0
GPVersion	23	2	23	0	0	2	0	0	1	0	0	1	0	0	0	0	0	0
GanttDialogInfo	11	1	11	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0
Connector	3	4	3	0	0	0	4	0	1	0	0	0	0	1	0	0	0	4
LoadDistribution	4	11	4	0	0	6	5	0	1	0	0	0.55	0.45	0	0	0	2	7
CustomColumEvent	8	7	7	0	1	6	1	0	0.88	0	0.12	0.86	0.14	0	0	0	1	3

Quality Metrics Values for Class Diagrams Post-refactoring

Class	After refactoring																	
	NumAttr	NumOps	numPubAttr	NumPriAttr	numProAttr	NumPubOps	numPriops	numProOps	ratioPubAttr	ratioPriAttr	ratioProAttr	ratioPubOps	ratioPriOps	ratioProOps	NOC	DIT	EC_Par	IC_Par
mxGraphView	3	12	0	1	2	10	0	2	0	0.33	0.67	0.83	0	0.17	0	1	0	4
mxGraphHandler	10	17	1	0	9	13	0	4	0.1	0	0.9	0.76	0	0.24	0	1	1	2
mxUtils	3	19	2	0	1	19	0	0	0.67	0	0.33	1	0	0	0	0	0	7
mxEdgeHandler	5	19	0	0	5	9	0	10	0	0	1	0.47	0	0.53	0	1	0	2
mxCellStatePreview	6	15	0	0	6	13	0	2	0	0	1	0.87	0	0.13	0	0	0	8
mxGraphics2DCanvas	6	19	2	0	4	15	0	4	0.33	0	0.67	0.79	0	0.21	0	1	0	9
mxCoordinateAssignment	14	12	0	1	13	12	0	0	0	0.07	0.93	1	0	0	0	0	0	1
mxCellEditor	19	18	3	3	13	17	0	1	0.15	0.15	0.7	0.94	0	0.06	0	0	0	1
mxMovePreview	10	15	0	0	10	14	0	1	0	0	1	0.93	0	0.07	0	1	0	2
mxGraphTransferHandler	8	15	1	1	6	13	0	2	0.13	0.13	0.74	0.87	0	0.13	0	0	0	1
mxConnectionHandler	12	18	1	1	10	18	0	0	0.08	0.08	0.84	1	0	0	0	1	0	2
mxGraphHierarchyModel	6	9	3	1	2	6	0	3	0.5	0.16	0.33	0.67	0	0.33	0	0	0	5
mxGraphOutline	12	12	1	1	10	12	0	0	0.08	0.08	0.84	1	0	0	0	0	0	0
mxSelectionCellsHandler	5	10	1	1	3	10	0	0	0.2	0.2	0.6	1	0	0	0	0	0	1
mxCellMarker	7	18	1	1	5	15	0	3	0.14	0.14	0.72	0.83	0	0.17	0	0	0	8
mxGraphStructure	2	17	0	2	0	16	0	1	0	1	0	0.94	0	0.06	0	0	0	11
mxGraphModel	2	19	0	0	2	15	0	4	0	0	1	0.79	0	0.21	0	1	0	5
mxCellHandler	4	15	0	0	4	12	0	3	0	0	1	0.8	0	0.2	0	0	0	1
BasicGraphEditor	6	15	0	1	5	10	0	5	0	0.16	0.84	0.67	0	0.33	4	0	15	3
mxFastOrganicLayout	15	17	0	0	15	14	0	3	0	0	1	0.82	0	0.18	0	1	0	1
ClassMap	3	5	0	3	0	5	0	0	0	1	0	1	0	0	0	0	1	0
InstrumentTask	5	14	0	5	0	10	0	4	0	1	0	0.71	0	0.29	0	1	0	0
HTMLReport	3	10	0	3	0	1	9	0	0	1	0	0.1	0.9	0	0	0	0	4
GanttProject	8	18	0	6	2	17	0	1	0	0.75	0.25	0.94	0.06	0	0	1	5	8
TaskManagerImpl	6	19	0	6	0	13	4	2	0	1	0	0.68	0.21	0.11	0	0	2	16
ChartModelBase	10	19	1	8	1	14	1	3	0.1	0.8	0.1	0.74	0.05	0.16	2	0	4	10
GanttOptions	8	19	1	7	0	17	2	0	0.13	0.88	0	0.89	0.11	0	0	1	1	4
UIFacadeImpl	7	19	0	7	0	14	4	1	0	1	0	0.74	0.21	0.05	0	0	1	9
XMLElement	4	19	1	3	0	13	0	6	0.25	0.75	0	0.68	0	0.32	0	0	1	3
DefaultDrawingView	5	19	0	5	0	15	1	3	0	1	0	0.79	0.05	0.16	0	0	0	6
EditorMenuBar	1	2	0	1	0	2	0	0	0	1	0	1	0	0	0	0	0	2
mxVmlCanvas	1	9	0	0	1	9	0	0	0	0	1	1	0	0	0	1	0	2
mxHtmlCanvas	1	10	0	0	1	9	0	1	0	0	1	0.9	0	0.1	0	1	0	2
SetLabelPositionAction	2	2	0	0	2	2	0	0	0	0	1	1	0	0	0	0	0	0
mxChildChangeCodec	0	6	0	0	0	6	0	0	0	0	0	1	0	0	0	1	0	8
CodeInstrumentationTask	4	7	0	4	0	0	7	0	0	1	0	0	1	0	0	0	1	5
Cobertura	7	8	0	7	0	6	2	0	0	1	0	0.75	0.25	0	0	0	0	2
ReportMain	1	5	0	1	0	2	3	0	0	1	0	0.4	0.6	0	0	0	0	0
HTMLReportFormatStrategy	1	1	0	1	0	1	0	0	0	1	0	1	0	0	0	0	0	0
OverwritingMerger	2	5	0	2	0	3	2	0	0	1	0	0.6	0.4	0	0	0	0	6
DayGridSceneBuilder	4	9	0	4	0	2	7	0	0	1	0	0.22	0.78	0	0	1	0	5
CalendarSaver	2	1	0	2	0	0	1	0	0	1	0	0	1	0	0	1	0	0
VerticalLayouter	0	1	0	0	0	1	0	0	0	0	0	1	0	0	0	1	0	0
HorizontalLayouter	0	1	0	0	0	1	0	0	0	0	0	1	0	0	0	1	0	0
NodeFigure	5	9	0	5	0	7	1	1	0	1	0	0.78	0.11	0.11	0	3	0	5
mxConstants	188	7	0	188	0	6	0	1	0	1	0	0.86	0	0.14	0	0	0	3
mxSwingConstants	17	2	0	17	0	2	0	0	0	1	0	1	0	0	0	0	0	0
mxGeometry	8	28	0	8	0	26	0	2	0	1	0	0.93	0	0.07	0	2	0	14
mxPngSuggestedPaletteEntry	8	0	0	8	0	0	0	0	0	1	0	0	0	0	0	0	0	0
ParseException	3	7	0	3	0	7	0	0	0	1	0	1	0	0	0	0	0	0
mxImage	4	7	0	4	0	7	0	0	0	1	0	1	0	0	0	0	0	0
mxEvent	57	0	0	57	0	0	0	0	0	1	0	0	0	0	0	0	0	0
mxEdgeStyle	29	0	0	29	0	0	0	0	0	1	0	0	0	0	0	0	0	0
AbstractCoberturaTestCase	6	7	0	6	0	7	0	0	0	1	0	1	0	0	12	0	0	0
ObjectMetric	3	2	0	3	0	2	0	0	0	1	0	1	0	0	0	1	0	0
GPVersion	23	2	0	23	0	2	0	0	0	1	0	1	0	0	0	0	0	0
GanttDialogInfo	11	1	0	11	0	1	0	0	0	1	0	1	0	0	0	0	0	0
Connector	3	4	0	3	0	4	0	0	0	1	0	1	0	0	0	0	0	4
LoadDistribution	4	11	0	4	0	11	0	0	0	1	0	1	0	0	0	0	2	7
CustomColumEvent	8	7	0	8	0	7	0	0	0	1	0	1	0	0	0	0	1	3

REFERENCES

[1] G. Booch, J. Rumbaugh, and I. Jacobson, "The unified modeling language," *Unix Review,* vol. 14, p. 5, 1996.

[2] I. Gorton, *Essential software architecture*: Springer Science & Business Media, 2006.

[3] M. Fowler, *Refactoring: improving the design of existing code*: Pearson Education India, 1999.

[4] W. C. Wake, *Refactoring workbook*: Addison-Wesley Professional, 2004.

[5] M. Zhang, T. Hall, and N. Baddoo, "Code bad smells: a review of current knowledge," *Journal of Software Maintenance and Evolution: research and practice,* vol. 23, pp. 179-202, 2011.

[6] M. Misbhauddin and M. Alshayeb, "UML model refactoring: a systematic literature review," *Empirical Software Engineering,* vol. 20, pp. 206-251, 2013.

[7] A. Ghannem, M. Kessentini, and G. El Boussaidi, "Detecting model refactoring opportunities using heuristic search," in *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research,* 2011, pp. 175-187.

[8] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum, "Maintainability defects detection and correction: a multi-objective approach," *Automated Software Engineering,* vol. 20, pp. 47-79, 2013.

[9] A. Ghannem, G. El Boussaidi, and M. Kessentini, "On the use of design defect examples to detect model refactoring opportunities," *Software Quality Journal,* pp. 1-19, 2013.

[10] J. Jürjens, *Secure systems development with UML*: Springer Science & Business Media, 2005.

[11] B. Alshammari, C. Fidge, and D. Corney, "Security metrics for object-oriented designs," in *Software Engineering Conference (ASWEC), 2010 21st Australian*, 2010, pp. 55-64.

[12] B. Alshammari, C. Fidge, and D. Corney, "Security metrics for object-oriented class designs," in *Quality Software, 2009. QSIC'09. 9th International Conference on,* 2009, pp. 11-20.

[13] B. Alshammari, C. Fidge, and D. Corney, "Assessing the impact of refactoring on security-critical object-oriented designs," in *Software Engineering Conference (APSEC), 2010 17th Asia Pacific,* 2010, pp. 186-195.

[14] M. El-Attar and J. Miller, "Improving the quality of use case models using antipatterns," *Software & Systems Modeling,* vol. 9, pp. 141-160, 2010.

[15] M. El-Attar and J. Miller, "Constructing high quality use case models: a systematic review of current practices," *Requirements Engineering,* vol. 17, pp. 187-201, 2012.

[16] Y. A. Khan and M. El-Attar, "Using model transformation to refactor use case models based on antipatterns," *Information Systems Frontiers,* pp. 1-34, 2014.

[17] A. Jedlitschka, M. Ciolkowski, and D. Pfahl, "Reporting experiments in software engineering," in *Guide to advanced empirical software engineering*, ed: Springer, 2008, pp. 201-228.

[18] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified Modeling Language Reference Manual, The*: Pearson Higher Education, 2004.

[19] O. M. GROUP, "UML 2.0 OCL Specification," ed: OMG Dokument ptc/2005-06-06.

[20] (2016, March, 5). *Design and UML class diagram*. Available: http://www.yyu.edu.tr/abis/admin/dosya/5637/files/UML%20Tasar%C4%B1m.pdf

[21] (2016, March, 5). *UML 2 Sequence Diagrams: An Agile Introduction*. Available: http://www.agilemodeling.com/artifacts/sequenceDiagram.htm

[22] (2016, March, 5). *UML Use Case*. Available: http://www.uml-diagrams.org/examples/hospital-management-use-case-diagram-example.html?context=uc-examples

[23] R. Hurlbut, "A survey of approaches for describing and formalizing use cases," *Expertech, Ltd*, 1997.

[24] N. Bevan, "Measuring usability as quality of use," *Software Quality Journal*, vol. 4, pp. 115-130, 1995.

[25] M. Genero, J. Olivas, M. Piattini, and F. Romero, "Using metrics to predict OO information systems maintainability," in *Advanced Information Systems Engineering*, 2001, pp. 388-401.

[26] R. Marinescu, "Measurement and quality in object-oriented design," in *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, 2005, pp. 701-704.

[27] C. NIST, "Glossary of Key Information Security Terms," ed: National Institute of Standards and Technology Gaithersburg, MD, 2006.

[28] M. Whitman and H. Mattord, *Principles of information security*: Cengage Learning, 2011.

[29] G. Suryanarayana, G. Samarthyam, and T. Sharma, "Chapter 2 - Design Smells," in *Refactoring for Software Design Smells*, G. Suryanarayana and G. S. Sharma, Eds., ed Boston: Morgan Kaufmann, 2015, pp. 9-19.

[30] N. Fenton and J. Bieman, *Software metrics: a rigorous and practical approach*: CRC Press, 2014.

[31] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *Software Engineering, IEEE Transactions on*, vol. 20, pp. 476-493, 1994.

[32] T. v. Enckevort, "Refactoring UML models: using openarchitectureware to measure uml model quality and perform pattern matching on UML models with OCL queries," in *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, 2009, pp. 635-646.

[33] J. Al Dallal and L. C. Briand, "An object-oriented high-level design-based class cohesion metric," *Information and software technology*, vol. 52, pp. 1346-1361, 2010.

[34] I. H. Moghadam and M. O. Cinneide, "Automated refactoring using design differencing," in *Software maintenance and reengineering (CSMR), 2012 16th European conference on*, 2012, pp. 43-52.

[35] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *Software Engineering, IEEE Transactions on*, vol. 28, pp. 4-17, 2002.

[36] W. F. Opdyke, "Refactoring object-oriented frameworks," University of Illinois at Urbana-Champaign, 1992.

[37] T. Mens and T. Tourwé, "A survey of software refactoring," *Software Engineering, IEEE Transactions on,* vol. 30, pp. 126-139, 2004.

[38] D. B. Roberts and R. Johnson, *Practical analysis for refactoring*: University of Illinois at Urbana-Champaign, 1999.

[39] C. F. Lange and M. R. Chaudron, "Managing model quality in UML-based software development," in *Software Technology and Engineering Practice, 2005. 13th IEEE International Workshop on*, 2005, pp. 7-16.

[40] A. A. Jalbani, J. Grabowski, H. Neukirchen, and B. Zeiss, "Towards an integrated quality assessment and improvement approach for UML models," in *SDL 2009: Design for Motes and Mobiles*, ed: Springer, 2009, pp. 63-81.

[41] T. Yue, S. Ali, and M. Elaasar, "A framework for measuring quality of models: experiences from a series of controlled experiments," *Simula Research Laboratory, Oslo, Norway,* 2010.

[42] G. Spanoudakis and A. Zisman, "Inconsistency management in software engineering: Survey and open research issues," *Handbook of software engineering and knowledge engineering,* vol. 1, pp. 329-380, 2001.

[43] T. Massoni, "Introducing Refactoring to Heavyweight Software Processes," Technical Report, CIn-UFPE, Brasil2003.

[44] P. Bottoni, F. Parisi-Presicce, and G. Taentzer, "Coordinated distributed diagram transformation for software evolution," *Electronic Notes in Theoretical Computer Science,* vol. 72, pp. 59-70, 2003.

[45] P. Van Gorp, H. Stenten, T. Mens, and S. Demeyer, "Towards automating source-consistent UML refactorings," in *«UML» 2003-The Unified Modeling Language. Modeling Languages and Applications,* ed: Springer, 2003, pp. 144-158.

[46] A. Tsiolakis, "Consistency Analysis of UML Class and Sequence Diagrams based on Attributed Typed Graphs and their Transformation1," 2000.

[47] M. Boger, T. Sturm, and P. Fragemann, "Refactoring browser for UML," in *Objects, components, architectures, services, and applications for a networked world*, ed: Springer, 2002, pp. 366-377.

[48] J. Xu, W. Yu, K. Rui, and G. Butler, "Use case refactoring: a tool and a case study," in *Software Engineering Conference, 2004. 11th Asia-Pacific*, 2004, pp. 484-491.

[49] C. Jeanneret, L. Eyer, S. Marković, and T. Baar, "RoclET–refactoring OCL expressions by transformations," in *Software & Systems Engineering and their Applications, 19th International Conference, ICSSEA 2006*, 2006.

[50] InterlliJ IDEA. (2016, 23, Feb). Available: http://www.intellij.com/idea

[51] RefactorIt. (2016, 23, Feb). Available: http://www.refactorit.com

[52] JRefactory. (2016, 23, Feb). Available: http://jrefactory.sourceforge.net

[53] jFactor. (2016, 23, Feb). Available: http://www.instantiations.com/jfactor

[54] Martin Fowler. (2016, 24 Feb). Available: http://refactoring.com/

[55] W. H. Brown, R. C. Malveau, and T. J. Mowbray, "AntiPatterns: refactoring software, architectures, and projects in crisis," 1998.

[56] M. J. Munro, "Product metrics for automatic identification of" bad smell" design problems in java source-code," in *Software Metrics, 2005. 11th IEEE International Symposium*, 2005, pp. 15-15.

[57] W. H. Cushman and D. J. Rosenberg, "Human factors in product design," *Advances in human factors/ergonomics*, vol. 14, 1991.

[58] R. Shatnawi and W. Li, "An investigation of bad smells in object-oriented design," in *Information Technology: New Generations, 2006. ITNG 2006. Third International Conference on*, 2006, pp. 161-165.

[59] A. Monden, D. Nakae, T. Kamiya, S.-i. Sato, and K.-i. Matsumoto, "Software quality analysis by code clones in industrial legacy software," in *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on*, 2002, pp. 87-94.

[60] C. J. Kapser and M. W. Godfrey, "“Cloning considered harmful” considered harmful: patterns of cloning in software," *Empirical Software Engineering*, vol. 13, pp. 645-692, 2008.

[61] S. Counsell, R. M. Hierons, R. Najjar, G. Loizou, and Y. Hassoun, "The effectiveness of refactoring, based on a compatibility testing taxonomy and a dependency graph," in *Testing: Academic and Industrial Conference-Practice And Research Techniques, 2006. TAIC PART 2006. Proceedings*, 2006, pp. 181-192.

[62] J. Al Dallal, "Identifying refactoring opportunities in object-oriented code: A systematic literature review," *Information and Software Technology*, vol. 58, pp. 231-249, 2015.

[63] T. Arendt and G. Taentzer, "Uml model smells and model refactorings in early software development phases," *Universitä̈t Marburg,* 2010.

[64] J. L. Vivas, J. A. Montenegro, and J. López, "Towards a business process-driven framework for security engineering with the UML," in *Information Security*, ed: Springer, 2003, pp. 381-395.

[65] M. Siponen and R. Baskerville, "A new paradigm for adding security into IS development methods," in *Advances in information security management & small systems security*, ed: Springer, 2001, pp. 99-111.

[66] C. Artelsmair, W. Essmayr, P. Lang, R. Wagner, and E. Weippl, "CoSMo: an approach towards conceptual security modeling," in *Database and Expert Systems Applications*, 2002, pp. 557-566.

[67] E. B. Fernandez, "A Methodology for Secure Software Design," in *Software Engineering Research and Practice*, 2004, pp. 130-136.

[68] I. Chowdhury, B. Chan, and M. Zulkernine, "Security metrics for source code structures," in *Proceedings of the fourth international workshop on Software engineering for secure systems*, 2008, pp. 57-64.

[69] D. Bhalla and L. B. California State University, *Automatic Detection of Bad Smells in Java Code*: California State University, Long Beach, 2009.

[70] T. Arendt, F. Mantz, and G. Taentzer, "EMF refactor: specification and application of model refactorings within the Eclipse Modeling Framework," in *of the BENEVOL workshop*, 2010.

[71] R. Fourati, N. Bouassida, and H. B. Abdallah, "A metric-based approach for anti-pattern detection in uml designs," in *Computer and Information Science 2011*, ed: Springer, 2011, pp. 17-33.

[72] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "DECOR: A method for the specification and detection of code and design smells," *Software Engineering, IEEE Transactions on,* vol. 36, pp. 20-36, 2010.

[73] P. Van Gorp, H. Stenten, T. Mens, and S. Demeyer, "Formal UML Support for the semi-automatic Application of object-oriented Refactorings," in *University of Antwerp*, 2003.

[74] T. Ruhroth, H. Voigt, and H. Wehrheim, "Measure, diagnose, refactor: A formal quality cycle for software models," in *Software Engineering and Advanced Applications, 2009. SEAA'09. 35th Euromicro Conference on*, 2009, pp. 360-367.

[75] M. Saeki and H. Kaiya, "Model Metrics and Metrics of Model Transformations," in *The First Workshop on Quality in Modeling*, 2006, pp. 31-45.

[76] M. Mohamed, M. Romdhani, and K. Ghedira, "M-REFACTOR: A New Approach and Tool for Model Refactoring," *ARPN Journal of Systems and Software (July 2011),* 2011.

[77] A. C. Jensen and B. H. Cheng, "On the use of genetic programming for automated refactoring and the introduction of design patterns," in *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, 2010, pp. 1341-1348.

[78] M. Van Kempen, M. Chaudron, D. Kourie, and A. Boake, "Towards proving preservation of behaviour of refactoring of UML models," in *Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*, 2005, pp. 252-259.

[79] E. Song, R. B. France, D.-K. Kim, and S. Ghosh, "Using roles for pattern-based model refactoring," in *Proceedings of the Workshop on Critical Systems Development with UML (CSDUML'02)*, 2002.

[80] C. Bouhours, H. Leblanc, and C. Percebois, "Bad smells in design and design patterns," *Journal of Object Technology,* vol. 8, pp. 43--63, 2009.

[81] D.-K. Kim, "Software quality improvement via pattern-based model refactoring," in *High Assurance Systems Engineering Symposium, 2008. HASE 2008. 11th IEEE*, 2008, pp. 293-302.

[82] K. Rui and G. Butler, "Refactoring use case models: the metamodel," in *Proceedings of the 26th Australasian computer science conference-Volume 16*, 2003, pp. 301-308.

[83] Ł. Dobrzański and L. Kuźniarz, "An approach to refactoring of executable UML models," in *Proceedings of the 2006 ACM symposium on Applied computing*, 2006, pp. 1273-1279.

[84] M. T. Llano and R. Pooley, "UML specification and correction of object-oriented anti-patterns," in *Software Engineering Advances, 2009. ICSEA'09. Fourth International Conference on*, 2009, pp. 39-44.

[85] G. Sunyé, D. Pollet, Y. Le Traon, and J.-M. Jézéquel, "Refactoring UML models," in *≪ UML≫ 2001—The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, ed: Springer, 2001, pp. 134-148.

[86] Checkstyle. (2016, 23, Feb). Available: http://checkstyle.sourceforge.net/

[87] Decor. (2016, 23, Feb). Available: http://www.ptidej.net/download

[88] iPlasma. (2016, 23, Feb). Available: http://loose.upt.ro/iplasma/index.html

[89] inFusion. (2016, 23, Feb). Available: http://www.intooitus.com/inFusion.html

[90] JDeodorant. (2016, 23, 2016). Available: http://www.jdeodorant.com/

[91] PMD. (2016, 23, Feb). Available: http://pmd.sourceforge.net/

[92] Stench Blossom. (2016, 23, Feb). Available: https://github.com/DeveloperLiberationFront/refactoring-tools/wiki/Stench-Blossom

[93] (2016, 6 June). *SDMetrics*. Available: http://www.sdmetrics.com/

[94] D. E. Golberg, "Genetic algorithms in search, optimization, and machine learning," *Addion wesley*, vol. 1989, p. 102, 1989.

[95] V. R. B.-G. Caldiera and H. D. Rombach, "Goal question metric paradigm," *Encyclopedia of Software Engineering*, vol. 1, pp. 528-532, 1994.

[96] Cinergix. (2016, 10 June). *Creately*. Available: http://creately.com/

[97] S. Systems. *Enterprise Architect*. Available: http://www.sparxsystems.com/products/ea/

[98] Microsoft. (2010). *Visual Studio*. Available: https://www.visualstudio.com/

[99] M. Ó Cinnéide, L. Tratt, M. Harman, S. Counsell, and I. Hemati Moghadam, "Experimental assessment of software metrics using automated refactoring," in *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, 2012, pp. 49-58.

[100] (2016, 12 July). *GitHub*. Available: https://github.com/

[101] Microsoft. (14 June). *MS Excel*. Available: https://products.office.com/en-us/excel

VITAE

Name	:	Haris Mumtaz
Nationality	:	Pakistan
Date of Birth	:	30/03/1988
Email	:	harismumtaz@outlook.com
Address	:	Islamabad, Pakistan
Academic Background	:	Haris Mumtaz has acquired his BS in Software Engineering from Bahria University, Islamabad, Pakistan in 2012. He has completed his MS in Software Engineering from King Fahd University of Petroleum and Minerals, Saudi Arabia. His research interests reside in software quality, software refactoring, software security, software metrics, anti-patterns and empirical software engineering.