International Journal of Security and Its Applications
Vol. 14, No. 1 (2020), pp.31-48
http://dx.doi.org/10.33832/ijsia.2020.14.1.03

IMPROVING THE SECURITY QUALITY OF USE CASE MODELS THROUGH THE APPLICATION OF SOFTWARE REFACTORING USING GENETIC ALGORITHM

Haris Mumtaz[1], Mohammad Alshayeb[2*], Sajjad Mahmood[3] and Mahmood Niazi[4]

[1]*Electrical, Computer and Software Engineering Department, University of Auckland, New Zealand*
[2,3,4]*Information and Computer Science Department, King Fahd University of Petroleum and Minerals, Saudi Arabia*
[1]hmum126@aucklanduni.ac.nz, [2]alshayeb@kfupm.edu.sa,
[3]smahmood@kfupm.edu.sa, [4]mkniazi@kfupm.edu.sa

***Abstract*— Use case modelling is an industrial de-facto standard technique to express functional requirements. Security bad smells are design flaws that can potentially degrade the quality of software by affecting a system's ability to prevent malicious activities. The presence of security bad smells in a use case model is likely to propagate security vulnerabilities to other software artefacts. Therefore, the detection and refactoring of security bad smells in use case models is important for ensuring the overall quality of software systems. In this paper, we propose a genetic algorithm-based detection approach to detect security bad smells. A refactoring process is then applied to correct the security bad smells. Finally, the improvement to security is assessed through the statistical analysis of quality metrics. The practicality of the approach is demonstrated by applying it to a set of use case models. The results show that the proposed security bad smell detection and correction technique can significantly improve the quality of use case models.**

***Keywords*— Use Case Refactoring; Software Security; Software Metrics**

1. INTRODUCTION

The Unified Modelling Language (UML) is a widely used analysis and design language for developing object-oriented systems [1]. A use case (UC)-driven approach is used to textually document requirements in the form of use cases. A use case diagram is used to provide a visual summary of use cases, actors and their relationships [2]. Use case modelling is performed at the early stage of requirements elicitation, and any design defects in use case diagrams propagate to the later stages of software development. Therefore, it is crucial to develop quality use case models through the early detection of poor design decisions to improve the overall end-product quality. These poor design decisions are commonly referred to as "bad smells", and taking the necessary measures to remove the bad smells is called "refactoring" [3].

Security is one of the important quality attributes that reflects the ability of a system to prevent malicious actions and loss of information. A basic aim of secure software is to prevent unauthorized access and the modification of information. The fulfilment of security requirements at the requirements level is imperative to minimize the cost of addressing the

Received: May 19, 2019
Reviewed: March 19, 2020
Accepted: March 23, 2020

ISSN: 2207-9629 IJSIA

security issue at later stages of the software development life cycle.

Despite the importance of security in use case models, a relatively small number of studies have focused on identifying bad smells and the related refactoring opportunities in use case models [4]. These studies have worked on improving the quality of use case models in terms of correctness, consistency, analytical behaviour and understandability. To the best of our knowledge, there exists no study that investigates the impact of model refactoring on the security quality of use case models. In particular, there is a need to develop detection and correction techniques to identify and mitigate security bad smells from use case models.

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

- To propose a detection technique to identify security bad smells in use case models.
- To propose a correction technique to correct security bad smells in use case models.
- To empirically assess the security improvements in use case models as a result of refactoring.

Our proposed approach is beneficial in a way that the best solution is generated after searching the entire problem space using genetic algorithm. The generated detection rules are generalized for the investigated security bad smells. The generalized rules could be directly applied to use case models to capture the investigated security bad smells. Another contribution of this study is the improvement of security aspects of use case models through the application of refactoring, which is supported by manual and statistical analysis.

The rest of this paper is structured as follows: Section 2 presents the related work. Section 3 discusses the research methodology. Section 4 presents the details of the experiments, the results and the validation. Section 5 presents the threats to validity and finally, Section 6 concludes this paper and provides directions for future work.

2. RELATED WORK

This section provides a detailed discussion on studies related to the research. This section covers bad smells and their detection techniques for use case models.

2.1 USE CASE BAD SMELLS

Generally, defects in use cases are referred to as anti-patterns instead of bad smells. In this research, we are treating anti-patterns in use cases as bad smells. Suryanarayana et al. classified design smells into four main categories: Abstraction, Encapsulation, Modularization and Hierarchy [5]. Each classification with the corresponding design smells is listed in Table I. In each classification, a significant number of design smells are reported. For example, in abstraction, there is a design smell, “missing abstraction”, which emphasizes a compromise on the integrity of data. Similarly, in deficient encapsulation, the class attributes are likely to be exposed to outside classes. Suryanarayana et al. also suggested some appropriate sets of refactoring opportunities for each design smell and also their impact on quality attributes.

Table I. Classification of Design Smells [5]

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

In this paper, we selected three security bad smells (missing hierarchy, broken modularization and missing modularization) from this classification. The rationale for selecting these bad smells is their abundance in the use case models. The design bad smells that are considered in this research are briefly described below:

Missing hierarchy: This aims to explicitly manage variation in hierarchical behaviour, where a hierarchy could have been created for the expected behaviour. This bad smell violates the reliability and integrity of data such that the access of the actor to an extension use case is inappropriate and leads to unexpected behavioural execution. This poses a major threat of security by compromising the reliability and integrity of data. The related refactoring strategy to cope with this bad smell is removing the connection with the inappropriate hierarchy interface.

Broken modularization: In the context of use case modelling, this bad smell happens when functionalities are split across multiple use cases using the include relationship. If an inclusion use case is included by a single use case, that affects the confidentiality and integrity attributes of security because functionalities are split and would require accessing from associations between inclusion and included use cases. The appropriate refactoring to overcome this bad smell is moving the inclusion use case functionality to the included use case.

Missing modularization: These bad smells occurs when an extension use case extends more than one use case. In other words, the extension use case is missing modularization in terms of providing exceptional functionality to a single use case. This bad smell violates the reliability and correctness attributes of security. The functionality of an extension use case must be confined to a single use case, otherwise, it would be uncertain which optional functionality to execute for which use case. This confusion might lead to unreliable and incorrect execution of data, which is closely related to security. The most effective refactoring strategy to eradicate this bad smell is splitting the extension use case into multiple use cases, depending on the number of extension relationships.

A few studies have addressed the issue of bad smells in a use case diagram and its description [4], [6]. A major contribution to the study of anti-patterns in a use case diagram is provided by El-Attar and Miller [4]. Their objective was to improve the quality of a use case diagram and its description. They utilized anti-patterns to identify bad smells in use cases and refactor them to improve the quality of use cases. Their study influenced the quality of use cases in terms of correctness, consistency, analytical ability and understandability.

Table II depicts the investigated use case anti-patterns and their respective refactoring strategies. The bad smells considered in this research are also tagged in Table II.

Table II. Use Case Anti-Patterns and Corresponding Refactoring Techniques [4]

Anti-pattern	**Refactoring**
Accessing a generalized concrete use case	r1. Concrete to Abstract r2. Drop Actor-Generalized UC Association
Accessing an extension use case [missing hierarchy]	r3. Drop Actor-Extension UC Association [applied refactoring] r4. Directed Actor-Extension UC Association
Using extension/inclusion use cases to implement an abstract use case	r5. Abstract Extended UC to Concrete r6. Inclusion to Generalization
Functional Decomposition: using the include relationship [broken modularization]	r7. Drop Functional Decomposition r8. Drop Functional Decomposition having Inclusion [applied refactoring]
Functional Decomposition: using the extend relationship [missing modularization]	r9. Split Extension UC [applied refactoring] r10. Extension to Generalization

Multiple generalizations of a use case	r11. Generalization to Include
Use cases containing common and exceptional functionality	r12. Drop Inclusion r13. Drop Extension
Multiple actors associated with one use case	r14. Generalize Actors r15. Split UCs
An association between two actors	r16. Drop Actor-Actor Association
An association between use cases	r17. Drop UC-UC Association
An unassociated use case	r18. Drop Unassociated UC
Two actors with the same name	r19. Rename Actor
An actor associated with an unimplemented abstract use case	r20. Abstract to Concrete r21. Add Concrete UC

2.2 BAD SMELL DETECTION AND REFACTORING

The classification of the bad smell detection techniques which we consider in this research is presented by Misbhauddin and Alshayeb [7]. They classified three detection strategies namely: design patterns, software metrics and pre-defined rules [7]. A few studies have been proposed using design patterns to detect bad smells on class diagrams [8]. We are not aware of any study which uses design patterns to detect bad smells on the use case model. Detection in use case models is mainly achieved using rule-based techniques. Refactoring is the process of improving the structure of a model without changing its behaviour [3] and was introduced into source code by Opdyke [9]. The refactoring concept has been extended to UML models to benefit from its ability to improve software model quality

Rule-based techniques ensure the use of a specific template or standard rules to develop software artefacts. Many researchers used rule-based techniques to detect bad smells on class diagrams [8],[10]. In the context of this study, only a few researchers have used rule-based techniques to detect bad smells on use case diagrams. El-Attar and Miller provided a template to specify use cases and relevant descriptions [2]. They reported that their proposed template enhances consistency in use case diagrams and their descriptions. They presented a semi-automated technique based on anti-patterns which provides a framework for defining anti-patterns to provide remedies for common quality problems in use case diagrams. The technique. They claimed that the application of their proposed technique would result in the use case model being 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 [4]. They used the model transformation approach using OCL to detect and refactor use case diagrams.

Rui and Bulter described applying refactoring to use cases and formulated a meta-model for use case diagrams [11]. They extended the use case meta-model to include inclusion, extension, generalization, precedence, similarity and equivalence. Butler and Xu considered the concept of refactoring in use case modelling in the context of product lines [12]. Product line variability and evolution were used to refactor use case models. Yu et.al used refactoring rules based on episodes to improve the quality of use case models [13]. Tanhaei *et al.*, proposed a framework to refactor software product lines while keeping them consistent with the feature model [14]. Kim and Doh proposed five use case refactoring strategies including: equivalence, decomposition, generalization, delete and merge [15]. These refactoring rules are implemented in the context of service-oriented architectures. Refactoring has also been used to improve the quality of use case descriptions.

Scant research has been conducted in the area of detecting bad smells in use case models using model metrics. Arendt and Taentzer used model metrics and model smells to propose a detection process [16]. They presented a study which integrates two tools, EMF Smell and EMF Refactor, which enables the automatic detection and removal of model smells by applying the suggested refactoring in a class diagram and use case diagram. They

investigated only one bad smell (unused use case) related to use case modelling. The goal of EMF Smell is to identify a model smell against a meta-model and present them in an understandable way. EMF Refactor consists of three main components: code generation module, refactoring application module and EMF model refactoring suite.

A few observations can be made from the discussion of the aforementioned detection techniques. Model smells in use cases are detected using metrics and pre-defined rules. In the case of the metrics-based technique, only one bad smell is studied, while rule-based techniques investigate a large set of bad smells. It can also be observed that the related literature has not studied bad smells in use case models from a security point of view. Another concern to be noted is the lack of work investigating the impact of refactoring on the quality of use case models from a security perspective. These gaps in the literature motivate us to study bad smells from a security perspective and assess the impact of refactoring on quality improvement in use case models.

3. RESEARCH METHODOLOGY

This section highlights the research methodology used to detect, correct and evaluate security bad smells in use case models.

3.1 RESEARCH METHODOLOGY OVERVIEW

First, we identify security bad smells from the existing bad smell catalogues. The selected security bad smells, along with the relevant quality metrics, are used as input to the generic algorithm (GA). The GA produces a set of detection rules to automatically detect security vulnerabilities in use case models.

To undertake the correction process, the use case models are transformed into XML representations. Based on the detected security bad smells, the relevant refactoring strategies are applied to the XML representations of use case models, which results in the generation of refactored XML representations. The refactored XML representations are exported back to the corresponding refactored use case models. The refactored use case models are processed using post refactoring conditions to ensure behavioural preservation. The quality metrics are computed, before and after refactoring, using XML representations of the use case models. The comparison of quality metrics computed before and after refactoring assists in evaluating the quality improvement of the use case models from a security perspective. The basic flow of activities is shown in

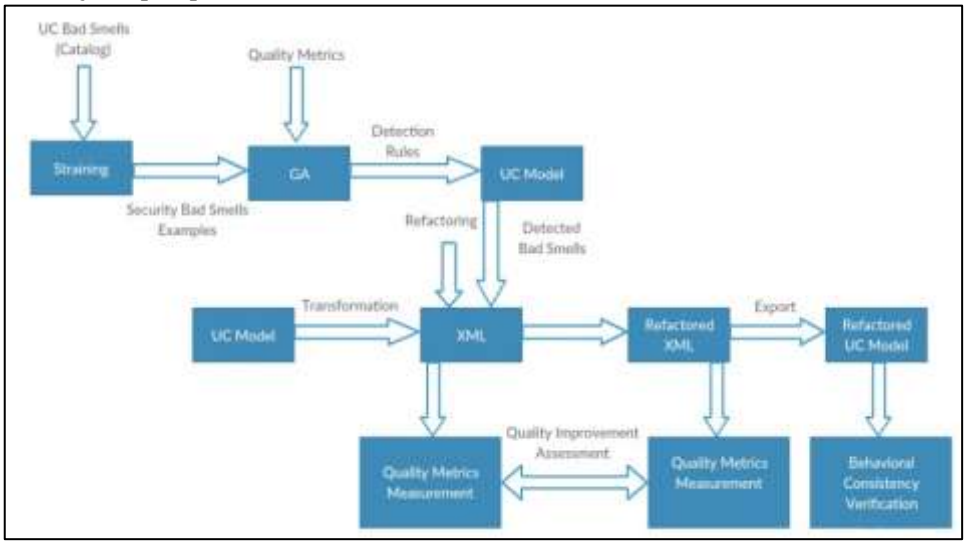

Fig. *1*.

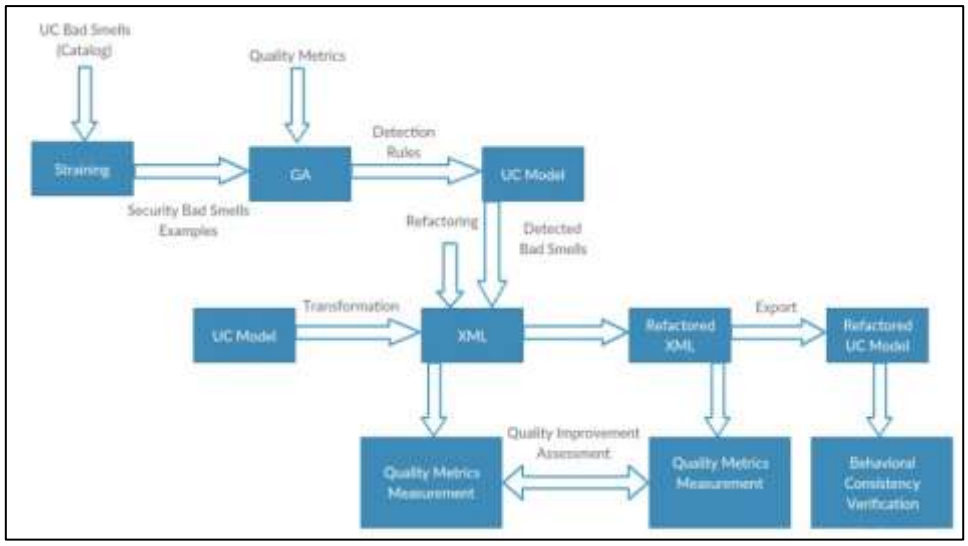

Fig. 1 Research Methodology Overview

3.2 SECURITY BAD SMELL SELECTION

The bad smells degrade software quality by compromising the quality attributes. A bad smell could impact one or more quality attributes. For example, Functional Decomposition is a bad smell that could impact maintainability of a software system, hence treated as a maintainability bad smell [10]. Similarly, other studies showed a subset of bad smells that could degrade maintainability of software system [17]. We adopted the design smell catalogue of Suryanarayana *et al.*, [5] to select security bad smells for use case models. In their catalogue, they reported the violation of several security attributes by design smells. If a bad smell violates any security attribute, it is considered a security bad smell. The existing catalogue of model bad smells is passed through this process to analyse which model bad smell affects the security attributes. For example, according to Suryanarayana et al., the missing modularization model smell affects understandability, changeability, extensibility, reusability, testability and reliability [5]. According to the security definitions, reliability is one of the security attributes [18]. Hence, missing modularization can be tagged as a security bad smell. Similarly, other security bad smells are filtered. The forthcoming example identifies a bad smell as a non-security bad smell using the same procedure. According to Suryanarayana *et al.*, imperative abstraction impacts understandability, changeability, extensibility, reusability and testability [5]. Since, none of these quality attributes is related to security as per the security definitions, this bad smell is not filtered by the process. In a similar manner, other non-security bad smells are identified. As a result of this filtration process, we selected three security bad smells: missing hierarchy, broken modularization and missing modularization [3], [5]. The definition of each security bad smell, along with the security requirements it violates, and appropriate refactoring, are available online[1].

3.3 SECURITY BAD SMELL DETECTION

The gaps identified in the literature are addressed by the proposed detection and correction approaches. The detection approach uses the approach proposed by Ouni *et al.*, [10], however, this study implements changes in the GA process, specifically for crossover and mutation operations in addition to using security bad smells instead of normal class level bad smells. Studying a different set of bad smells enforces the use of a different set of quality metrics. In addition, we apply it to use case diagrams.

[1] http://softwareengineeringresearch.net/UCSBS/TaxonomyOfSecurityBadSmells.pdf

The GA [10] is used to generate detection rules by using security bad smell examples and relevant quality metrics. The quality metrics are collected using the SDMetrics tool [19]. The near optimal solution obtained from the GA presents the rules that detects the maximum number of security bad smells in UC models. Initially, the experiment is performed with a small dataset of use case models, then, to justify the use of GA, experiments with large datasets are conducted. The initial experiments would also contribute to the understandability of the detection approach. The pseudo-code of the genetic algorithm is presented in Fig. 2 [10]. The algorithm takes quality metrics and bad smell examples as inputs to produce a set of detection rules to detect security bad smells in UC models. The algorithm stops when the maximum number of the security bad smells are detected.

Individuals are composed of three rules; each of these rules specifies a different bad smell for missing hierarchy, broken modularization and missing modularization, as shown in Fig. 3. The security bad smell is detected when the metric meets the threshold conditions specified in the individual rule.

```
Input:
Quality metrics
Bad smell examples
Process:
1.  I = set of rules
2.  P = set of I
3.  U = use case model
4.  repeat
5.      for all I in P do
6.          detected bad smells = execute_rules(U);
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
```

Fig. 2 A high-level GA adaptation for detection [10]

R1: IF (isExtension (u) == true AND NumAss (u) >= 2) THEN missing hierarchy
R2: IF (isExtension (u) == true AND extending (u) >= 4) THEN missing modularization
R3: IF (isInclusion (u) == true AND included (u) == 2) THEN broken modularization

Fig. 3 Individual representation

For crossover and mutation, the individuals are selected based on their relevant fitness value. The fitness value is calculated for each individual in each iteration. Two thirds of the best-fitted individuals are selected while the remaining one-third are discarded. The discarded population is regenerated from the selected ones through crossover and mutation. For crossover, one of the three rules is randomly selected and swapped with the same in another individual to create two new individuals. The mutation is achieved by changing quality metric values by randomly selecting an individual and corresponding rule. Depending on the generated rule, metric values can either be increased or decreased by one.

The fitness function calculates the number of detected bad smells compared to the total number of existing bad smells in the UC model. The fitness value is incremented by one when a rule is able to detect a security bad smell. The mathematical representation is shown

as follows:

$$f(I) = \sum_{i=1}^{D} (s_i) \quad (1)$$

where *f(I)* computes the fitness of an individual, *D* represents the number of defects in the use case diagrams, and s_i has a value of 1 if the *i*th defect is detected by the rule, and value of 0 if otherwise. The individuals with the highest fitness value are selected for crossover and mutation operations. Maximum fitness is achieved when the individuals are able to detect all the bad smells present in the model.

3.4 SECURITY BAD SMELL CORRECTION

The correction of security bad smells is achieved through model transformation. The use case diagrams are first transformed into XML, then quality metrics are extracted. The XML representation of the use case diagrams is corrected based on the refactoring of the identified security bad smells. The detected security bad smells in a use case model can easily be traced in the corresponding XML representation. Once refactoring is successfully applied, the corrected XML representations are exported back to the corresponding use case models. In this way, the use case models no longer contain security bad smells.

To maintain consistency between the original and the refactored models, we use post-conditions which are validated after applying refactoring. The presence of few post conditions is ensured for each security bad smell. The problem of missing hierarchy is itself an incorrect behaviour instance, so the removal of it makes the use case diagram behaviourally 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 behaviour 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 behavioural consistency is the presence of inclusion functionality. Whenever an 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 behaviour. 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 behavioural confined for a use case. The breaking of exceptional functionalities for each extended use case allows correct behaviour. The refactoring does not only ensure the consistency but also it ensures the behavioural correctness.

3.5 SECURITY IMPROVEMENT VALIDATION

We use metrics to assess the security improvement in the use case models as a result of refactoring. We measure the metrics before and after refactoring to assess the change in the metric values. We apply a pair-wise t-test to provide statistical evidence of the impact of the changed metrics.

4. THE EXPERIMENT

This section provides details on the implementation of the detection and correction approaches on use case models. The section also explains the experimental setup, results and validation. We use the guidelines in Jeditschka *et al.*, [20] to present the empirical validation.

4.1 EXPERIMENTAL GOALS

The research goal, using the Goal Question Metric (GQM) approach [21], is as follows: "*To analyse the model refactoring to security bad smells for the purpose of improving the quality of use case models with respect to security".*

The main goal is divided into multiple sub-goals. The sub-goals include the successful detection and correction of security bad smells and to what extent refactoring can improve the software in terms of security. The research questions to be addressed are as follows:

RQ1: To what extent can the proposed detection approach detect security bad smells in use case models?

RQ2: To what extent can the correction to security remove bad smells in use case models?

RQ3: To what extent can refactoring security bad smells improve the security aspects of use case models?

4.2 EXPERIMENTAL MATERIALS

Four use case models belonging to four different systems are used in the experiments [22]. The descriptive statistics of the four use case diagrams are presented in Table III. The statistics are presented in terms of the number of use cases present in the system, the number of actors interacting with the system, the number of includes relationships and the number of extends relationships. However, due to space limitation, we show the details and the results of only two case studies.

Table III. Statistics on the Investigated Use Case Diagrams

System	Use cases	Actors	Includes	Extends
ATM system (Fig. 4**Error! Reference source not found.**)	10	3	3	3
HR system (Fig. 5)	8	4	3	4
Restaurant system	13	4	1	10
Travel agency system	9	5	1	6

The security bad smells investigated in the use case diagram are: missing hierarchy, broken modularization and missing modularization. Multiple instances (total 27) of these three security bad smells are present in the use case diagrams.

Fig. 4 shows the use case diagram of an ATM system [22]. 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, all of which have generalization relationships. Transactions and system maintenance require the actor's login thus the include relationship is present. Both of these use cases can lead to the exceptional execution of the "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: The actor "Bank" is accessing the "System Reporting" extension use case. The "System Reporting" is executed by "Bank" without the knowledge of the "System Maintenance", which is the primary use case. The "System Reporting" would require some information from "System Maintenance" to correctly execute the service. This bad smell would lead to improper execution of the extension use case that may affect the reliability and integrity of data, which are necessary attributes of security.

Missing modularization:

UB2: The “Bad Pin” use case is extending two use cases: “System Maintenance” and “Transaction”.
An exceptional functionality should be executed for a single primary use case. The execution of “Bad Pin” might lead to incorrect initiation (violating reliability and correctness attributes of security) of respective exceptional services for “System Maintenance” and “Transaction” respectively.

Broken modularization:

UB3: “System Shutdown” is included in only one use case “System Maintenance”.
The unnecessary functional decomposition of a service would require information flow and communication between the decomposed parts. The information flow between “System Shutdown” and “System Maintenance” would impact the security attributes like confidentiality and integrity of data.

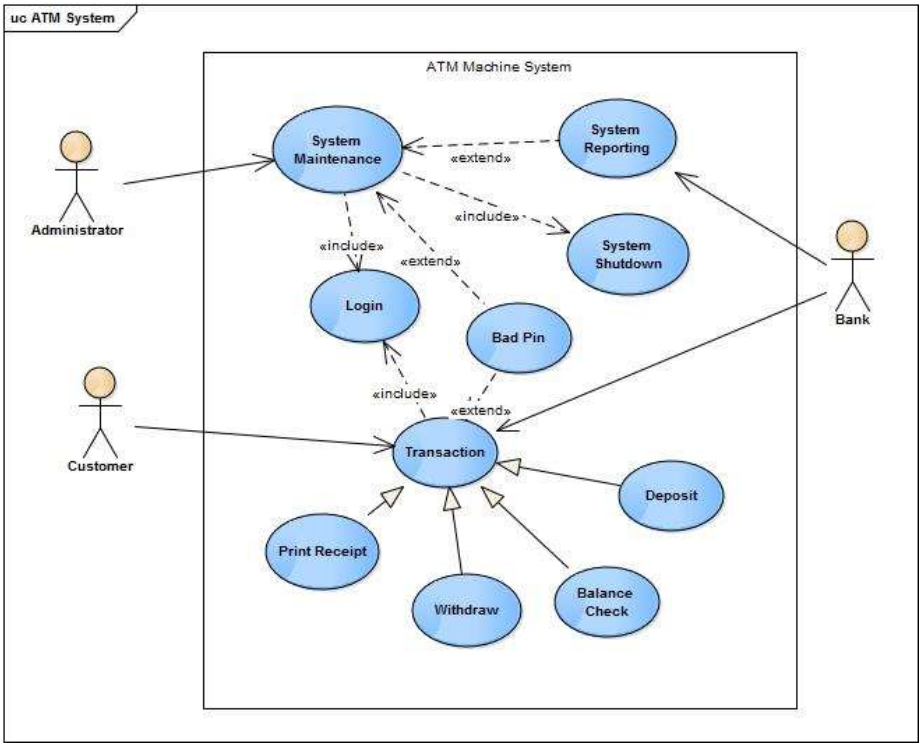

Fig. 4 Use Case Diagram of an ATM System

The use case diagram of a human resource system is shown in Fig. 5 [22]. It provides multiple services such as update benefits, elect reimbursement from health care, elect stock purchase and issue purchase invoice. In addition to receiving benefits updates, the employee has the option of electing reimbursement and stock purchases. The stock entity is responsible for electing and purchasing stock. The purchase invoice for stock is issued by a HR representative. The 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: The actor “Health Care Dept.” is accessing the “Elect Reimbursement from Health Care” extension use case.
UB5: The actor “Stock Entity” is accessing the “Elect Stock Purchase” extension use case.

Missing modularization:

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

Broken modularization:

UB7: The “Update Dental Plan” is included in only one use case “Update Benefits”.
UB8: The “Update Insurance Plan” is included in only one use case “Update Benefits”.
UB9: The “Update Medical Plan” is included in only one use case “Update Benefits”.

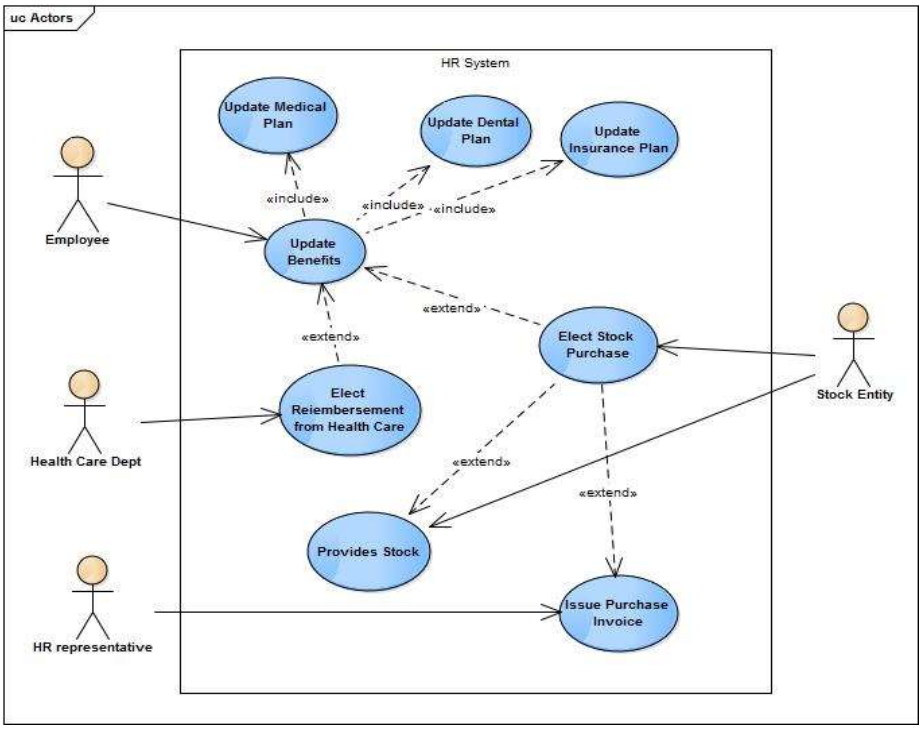

Fig. 5 Use Case Diagram of a HR System

The collection of quality metrics is accomplished using the SDMetrics tool [19]. The construction of the use case diagrams is performed using Enterprise Architect (EA). EA is used for model transformation from a use case to XML. Visual Studio is utilized to implement the GA.

4.3 VARIABLES

The dependent variable in this experiment is model quality. The independent variables are security bad smells and quality metrics.

We use recall for the security bad smells in the use case diagrams as the measure for the independent variable in the detection phase. For the correction phase, the independent variable is the correction efficacy in terms of removal percentage of security bad smells. For a quantitative assessment of the security improvement, quality metrics are used for the independent variable. The quality metrics selected for the use case diagrams are as follows:

- ***NumAss:*** Number of associations between a use case and actor(s).
- ***Including:*** The number of use case(s) a use case includes.
- ***Included:*** The number of use case(s) included by an inclusion use case.
- ***Extended:*** The number of use case(s) extended by an extension use case.
- ***Extending:*** The number of use case(s) a use case extends.

4.4 RESEARCH HYPOTHESES

The following hypotheses are formulated to statistically validate the effectiveness of the proposed approaches:

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 the detection technique is significant in terms of identifying the security bad smells in the investigated use case diagrams. The quantification of the hypothesis is necessary for later testing and is presented as follows 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 the correction technique is significant in terms of removing the security bad smells in the investigated use case diagrams. The quantification of the hypothesis is necessary for later testing and is presented as follows in terms of correction efficacy:

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

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

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

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

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

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

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

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

4.5 EXPERIMENT TASKS

Detection: The initial individuals are formed from existing security bad smells in the four use case diagrams. The aggregation of the individuals creates the initial population. The population undergoes selection, crossover and mutation operations. Once the GA reaches its terminating condition, it yields a solution carrying best fitness. The selection of the quality metrics and the formation of the rules are accomplished through the measurement of the metrics before and after refactoring.

Correction: The corrections in use case diagrams are accomplished by applying relevant refactoring techniques to the identified security bad smells. The mapping of the listed anti-patterns to security bad smells is based on their descriptions and violation of the security aspects. The correction approach uses model transformation using XMI for refactoring purposes. The 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 the ATM system (Fig. 4), there exists a security bad smell "missing hierarchy", where the actor "Bank" is accessing the "System Reporting" extension use case. This smell is corrected by removing the association between "Bank" and "System Reporting".

4.6 RESULTS

Detection: The detection rules generated by the GA are shown in Fig. 6. R1 measures the missing hierarchy security bad smell by using two conditional statements: *isExtension* and *NumAss* variables. For example, in Fig. 4, there exists an association between the "System Reporting" extension use case and the "Bank" actor, which is an instance of the missing hierarchy security bad smell. Similarly, R2 detects the missing modularization security bad smell. This rule also uses two conditional statements with variables: *isExtension* and *extending*. For example, in Fig. 4, the "Bad Pin" use case is *extending* two use cases, which is a clear missing modularization problem. Finally, R3 detects the broken modularization security bad smell. It uses two conditional statements with variables: *isInclusion* and *included*. For example, in Fig. 4, the "System Shutdown" use case is included by a use case.

R1: IF (isExtension (u) == true AND NumAss (u) >= 1) THEN missing hierarchy
R2: IF (isExtension (u) == true AND extending (u) >= 2) THEN missing modularization
R3: IF (isInclusion (u) == true AND included (u) == 1) THEN broken modularization

Fig. 6 Best solution generated for the use case diagrams

The set of rules is applied on the use case diagrams to evaluate the recall efficiency. The set of rules governing the best solution is able to identify all 27 security bad smells present in the four use case diagrams, meaning, the detection approach has 100% recall.

Correction: The correction approach is able to remove all the security bad smells in the use case diagrams. Table IV and Table V summarize the refactoring application to correct the security bad smells in the ATM, human resource, restaurant and travel agency systems, respectively. The resulting use case diagrams of the ATM, Human Resource, Restaurant and Travel Agency Systems after refactoring are shown Fig. 7 and Fig. 8 respectively. It can be observed from the refactored diagrams that the identified security bad smells have been removed.

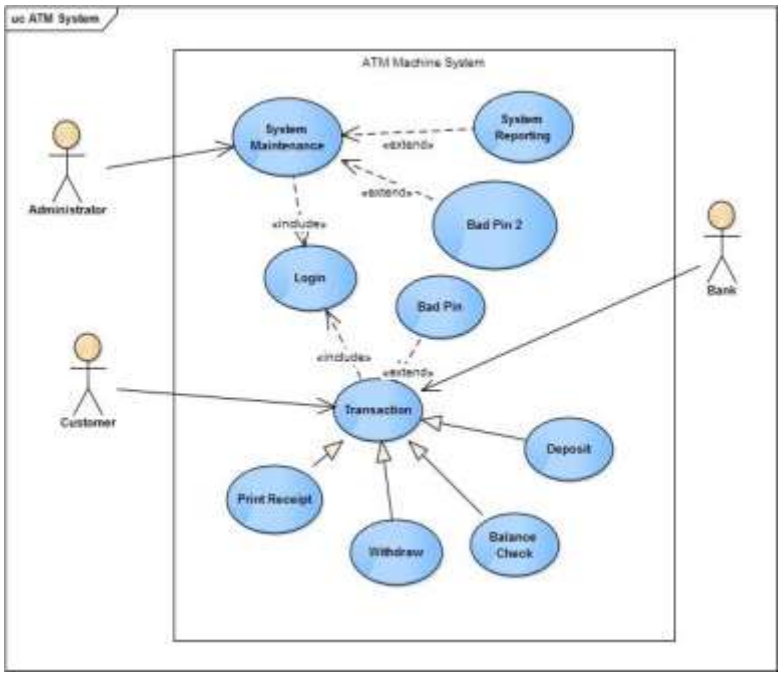

Fig. 7 Refactored Use Case Diagram of an ATM System

Table IV. Applied Refactoring in an 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

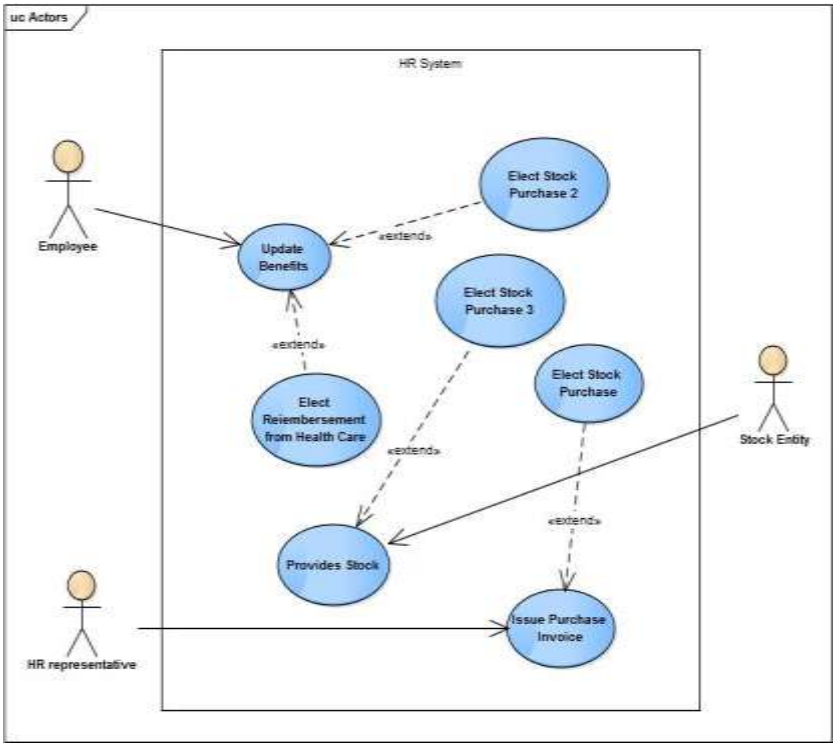

Fig. 8 Refactored Use Case Diagram of an HR System

Table V. Applied Refactoring in an HR System

Security bad smell ID	Applied refactoring
UB4	Drop association between Health Care Dept. and Elect Reimbursement from Health Care Dept.
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

4.7 HYPOTHESES TESTING

This section discusses the testing of the research hypotheses.

Hypothesis 1 (RQ1): In order to test hypothesis 1, the Detection Recall (DR) is calculated. The null hypothesis (H_{01}) can be rejected if DR is significant. In numerical terms, if DR is greater than or equal to 80%, the null hypothesis (H_{01}) is rejected. The detection approach shows a significant DR of 100% while being executed on the use case diagrams. The DR is greater than 80%, so null hypothesis (H_{01}) is rejected. This answers RQ1 that the proposed detection approach is able to detect 100% of the security bad smells in use case diagrams.

Hypothesis 2 (RQ2): To test hypothesis 2, the Correction Efficacy (CE) 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}) is rejected. The correction approach shows 100% of the security bad smells in the use case diagrams are detected. The CE is greater than 80%, so null hypothesis (H_{01}) is rejected. This addresses RQ2 that the proposed correction approach is able to remove 100% of the security bad smells in use case diagrams.

Hypothesis 3 (RQ3): The pair-wise t-test is used to empirically evaluate the impact of security bad smells in the use case models. The p-value is computed with 95% confidence through a pair-wise t-test. It is observed that the computed p-value is 0.0001, which is less than 0.05. This proves there has been significant security improvement in the use case diagrams and subsequently, answers RQ3. As a result, we reject the null hypothesis (H_{03}) with 95% confidence, hence, the sub-goal of security improvement in use case diagrams is achieved. The quality metric values before and after refactoring are provided online[2].

4.8 EXPERIMENTS WITH LARGE DATASETS

To demonstrate further confidence in the detection approach in terms of generating the set of rules, we conducted an experiment with a bigger data set. The supplementary experiments address the notion of result generalization. This data is generated in two ways: 1) simple replication; and 2) varied replication.

1) Experiments with Simple Replication Datasets

In simple replication, data is produced by replicating the small datasets.

Table VI shows the statistics of the replicated datasets. It can be seen that data size is significantly enlarged in terms of the number of use cases and security bad smell instances. Since the data size is increased, the number of security bad smell instances is also increased. The detection rules and DR remain unchanged after performing the experiment with the use case diagrams dataset.

[2] http://softwareengineeringresearch.net/UCSBS/QualityMetricsValuesBeforeAndAfterRefactoring.pdf

Table VI. Statistics of Simple Replicated Datasets

UML model	Number of use cases	Security bad smell instances
Use case diagram	1160	638

2) Experiments with Varied Replication Datasets

Varied replication is used to introduce more quality metric values. The dataset size is increased in terms of the number of use cases; hence the number of security bad smell instances is also increased. Table VII shows the statistics of the replicated datasets.

Table VII. Statistics of Replicated Datasets with Variations

UML model	Number of use cases	Security bad smell instances
Use case diagram	1280	704

When performing the supplementary experiments with varied replication datasets for use case diagrams, the detection rules remain same. The experiments with a small dataset, a simple replication dataset and a varied replication dataset produce the same detection rules and DR.

In the detection approach, the most important element is the examples of security bad smells because these are used to formulate the rules. We used examples of bad smells from online repositories which allow the actual programming practices in the detection process to be identified. The rules generation process is executed multiple times using bad smell examples to reduce any uncertainty with respect to the quality of the rules.

In order to improve the consistency of results, we performed experiments with large datasets of use case models. Although the experiments with small datasets give significant results, the experiments with the large datasets further strengthen the confidence in the consistency of the results, which can be observed from the achieved detection recall. The generated solution from multiple runs of GA yielded solutions with minimal difference in fitness, which means that the detection approach is stable. During correction, the XMI transformations from the use case models also remain consistent.

During the validation of the security improvement, changes in the quality metric values are observed. All the quality metrics contribute to the improvement in security in a specific use case model, but the impact of the metrics may vary depending on the security bad smell being removed. For instance, the refactoring of missing modularization results in significant changes to the metric values because the extension use case undergoes decomposition. On the other hand, refactoring broken modularization marginally changes the metrics.

Although we have analysed the effect of refactoring on the security improvement, the use case models have shown improvements in other quality properties as well. A notable enhancement in quality is observed in terms of modularity, complexity, reusability and design size. The refactoring of broken modularization in use case diagrams reduces the number of inclusion use cases, which suggests the model is less complex. The refactoring of inclusion use cases also reduces the quantity of use cases per actor, which means that the model size is decreased. The refactoring of missing modularization bad smells introduces modularity in the use case diagrams. The increased modularity in use case diagrams allows reduced complexity and enhanced reusability.

5. THREATS TO VALIDITY

One threat to validity that may affect this research is the selection of the independent variables. However, in the 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 use case models. Another construct

validity threat is concerned with security bad smell examples. Some security bad smell examples might be overlooked during individual formulation; however, this threat is mitigated by crossover and mutation operations.

The replication of datasets is performed manually so it may cause threats to the conclusion validity. This threat is minimized by making the replication random. Randomization introduces diversity in the datasets, which is the ultimate objective, regardless of the manual replication.

An external threat validity is related to the generalization of the results which is concerned with the selected use case models. However, we used four different use case models from different domains. In addition, we used large datasets to improve the generalization of the results.

6. CONCLUSION AND FUTURE WORK

In this research, we overcome the problem of security in use case models by applying software refactoring. The detection of security bad smells is achieved through the adaptation of the genetic algorithm approach, while the correction is accomplished by the model transformation approach. The detection approach uses quality metrics to formulate rules. The best set of rules generated by the GA is used for the detection of security bad smells in the use case models. The correction approach applies a model transformation to XML representation for refactoring the identified security bad smells in use case models.

The proposed approaches are validated by performing experiments with multiple use case models. The detection approach is able to detect security bad smells with 100% recall. The correction approach was also able to remove 100% of the security bad smells. We conducted supplementary experiments to generate more generalized detection rules because the detection approach relies heavily on generated rules. Although the sets of rules generated by the supplementary experiments remain the same, they enhance confidence in the generated rules. Through the statistical analysis of quality metrics, we were able to conclude that there is a significant improvement in the security quality of use case models.

The results of the detection and correction approaches motivated us to extend the work in the future. We plan to apply the approaches to other UML models to gain further confidence in their applicability to other models, such as sequence diagrams. We also plan to apply the same approaches with a different set of security bad smells to assess their appropriateness with other security bad smells.

ACKNOWLEDGMENT

The authors would like to acknowledge the support provided by the Deanship of Scientific Research at King Fahd University of Petroleum and Minerals, Saudi Arabia.

REFERENCES

[1] G. Booch, J. Rumbaugh, and I. Jacobson, "The unified modeling language," Unix Review, vol. 14, no. 13, p. 5, (1996).
[2] M. El-Attar and J. Miller, "Improving the quality of use case models using antipatterns," Software & Systems Modeling, vol. 9, no. 2, pp. 141-160, (2010).
[3] M. Fowler, K. Beck, J. Brant, and W. Opdyke, Refactoring: Improving the Design of Existing Code. Addison-Wesley, (1999).
[4] "Using model transformation to refactor use case models based on antipatterns," Information Systems Frontiers, pp. 1-34, (2014).
[5] G. Suryanarayana, G. Samarthyam, and T. Sharma, "Chapter 2 - Design Smells," in Refactoring for Software Design Smells, G. Suryanarayana and G. S. Sharma, Eds. Boston: Morgan Kaufmann, (2015), pp. 9-19.
[6] T. Arendt and G. Taentzer, "UML model smells and model refactorings in early software development phases," Universita¨t Marburg, (2010).
[7] M. Misbhauddin and M. Alshayeb, "UML model refactoring: a systematic literature review," Empirical Software Engineering, vol. 20, no. 1, pp. 206-251, (2013).

[8] 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).
[9] W. Opdyke, "Refactoring Object-Oriented Frameworks," PhD thesis PhD thesis, University of Illinois at Urbana Champaign, (1992).
[10] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum, "Maintainability defects detection and correction: a multi-objective approach," (in English), Automated Software Engineering, vol. 20, no. 1, pp. 47-79, 2013/03/01 (2013).
[11] K. Rui and G. Butler, "Refactoring use case models: the metamodel," in Proceedings of the 26th Australasian Computer Science Conference, (2003), vol. 16, pp. 301-308: Australian Computer Society, Inc.
[12] G. Butler and L. Xu, "Cascaded refactoring for framework," SIGSOFT Software Enggineering Notes, vol. 26, no. 3, pp. 51-57, (2001).
[13] W. Yu, J. Li, and G. Butler, "Refactoring Use Case Models on Episodes," in 19th IEEE International Conference on Automated Software Engineering (ASE'04), (2004), pp. 328-331.
[14] M. Tanhaei, J. Habibi, and S.-H. Mirian-Hosseinabadi, "A Feature Model Based Framework for Refactoring Software Product Line Architecture," Journal of Computer Science and Technology, journal article vol. 31, no. 5, pp. 951-986, September 01 (2016).
[15] Y. Kim and K.-G. Doh, "The Service Modeling Process Based on Use Case Refactoring," in Business Information Systems, vol. 4439, W. Abramowicz, Ed. (Lecture Notes in Computer Science, Berlin / Heidelberg: Springer, (2007), pp. 108-120.
[16] T. Arendt, F. Mantz, and G. Taentzer, "EMF refactor: specification and application of model refactorings within the Eclipse Modeling Framework," in the BENEVOL Workshop, (2010).
[17] D. Steidl and S. Eder, "Prioritizing maintainability defects based on refactoring recommendations," presented at the Proceedings of the 22nd International Conference on Program Comprehension, Hyderabad, India, (2014).
[18] M. Whitman and H. Mattord, Principles of information security. Cengage Learning, (2011).
[19] (2017, 6 June, 2017). SDMetrics. Available: http://www.sdmetrics.com/
[20] A. Jedlitschka, M. Ciolkowski, and D. Pfahl, "Reporting Experiments in Software Engineering," in Guide to Advanced Empirical Software Engineering, F. Shull, J. Singer, and D. I. K. Sjøberg, Eds. London: Springer London, (2008), pp. 201-228.
[21] V. R. B.-G. Caldiera and H. D. Rombach, "Goal question metric paradigm," Encyclopedia of Software Engineering, vol. 1, pp. 528-532, (1994).
[22] Cinergix. (2016, 10 June). Creately. Available: http://creately.com/

