A Novel Approach for Improving the Quality of Software Code using Reverse Engineering

Hamza A. Elghadhafi
Faculty of Information Technology
Benghazi University, Libya
hamza.elghadhafi@uob.edu.ly

Tawfig M. Abdelaziz
Faculty of Information Technology
Benghazi University, Libya
tawfig.tawuill@uob.edu.ly

Abdelsalam M. Maatuk
Faculty of Information Technology
Benghazi University, Libya
abdelsalam.maatuk@uob.edu.ly

ABSTRACT
Copying and pasting program code fragments with minor changes is a common practice in software development. Software systems often have similar segments of code, called code clones. Due to many reasons, unintentional smells may also appear in the source code without awareness of program developers. Code smell may violate the principles of software design and negatively impact program design quality, thus making software development and maintenance very costly. This paper presents an enhanced approach to facilitate the process of identification and elimination of code smells. The proposed solution allows the detection and removal of code smells for refinement and improvement of the quality of software system. Code smells are analyzed, restructured and eliminated from the source code using reverse engineering techniques. The solution maintains the external behaviour of software system and judges the efficiency of systems code. An experiment has been conducted using a real software system, which is evaluated before and after using the approach. The results have been encouraging and help in detecting code smells.

Keywords
Code Smells, Code Refactoring, Code Clone, Smells Detection.

1. INTRODUCTION
In software engineering, a software development life cycle (SDLC) is splitting software development process into distinct phases, containing activities with intent of better planning and management [13,19]. SDLC aims to produce a high-quality software that meets customer expectations and reach a product completion within time and cost [10, 20]. The implementation phase refers to the programs coding in SDLC, in which the final product is constructed successfully after different development stages [9,25, 26]. It includes program design and other related steps, such as code programming, documenting, testing, and bug fixing.

ICEMIS '18, June 19–21, 2018, Istanbul, Turkey © 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6392-1/18/04...$15.00
DOI: https://doi.org/10.1145/3234698.3234729

SDLC is an integrated process, which makes software construction easier, reduces the amount of problems encountered and tackles feedback that might affect the next phases [12]. For instance, the user requirements collected during the requirements phase are translated into architecture design, which is transformed into a working software code during the implementation phase. The software code is verified against the requirements during the testing phase. Then, the complete software product is delivered to customer in deployment phase. The actual problems and bugs that come up when the customers start using the software system are solved during maintenance phase [11].

In fact, the implementation phase has one key activity, i.e., writing code for a program. There are two different techniques for writing clean code, i.e., use brain (writing code from scratch) or copy-pasting technique. The copy-pasting technique is a popular for writing code, which is less time-consuming and cost [22]. However, using this technique might cause problems, which frequently appear in the source code. These problems are known as Code Smells, which is a surface indication corresponds to a deeper problem in the system [8].

A significant portion of the software development effort is spent on maintaining and improving existing systems rather than developing new ones. Having stated the fact that there has been a clear increase in the size of software systems, there is an increase in the size of system code too. This definitely makes the system code becomes more complex [7]. This complexity leads to unwelcome code smells to be exist in the source code, and eventually has a negative effect on the quality of systems [15, 21,22,23,24].

Software maintenance represents the costliest phase of the SDLC, where it is estimated to 40%-70% of the total cost of the project lifecycle. Consequently, reducing the effort spent on maintenance can be seen as a natural way of reducing the overall costs of a software project. This is one of the main reasons for the recent research in code smells. One of the most important aspects of the maintainability is code understandability, which makes the code easier to implement new segments and debugging the code [1]. It becomes very important to utilize approaches for detecting and removing code smells in order to refine and improve the quality of software system.

Due to requirements changing and growing the need for software improvement, upgrading legacy systems have become more complex and expensive tasks because of time-consuming process. Thus, there is a need for software engineering methods and tools that facilitate program understanding [2]. Generally, the need for maintaining existing software systems has become an important business goal in recent years in order to develop software efficiency, performance, maintainability, reusability and scalability [1, 17, 18].

This research focuses on the problems in the implementation and testing phases that have negative impacts on software code design, which lead to producing poor-quality software. This paper describes an approach that aims to improving the overall quality of software by eliminating the possibility of code problems. The proposed solution is considered as a guideline to tackle some of code issues. The main idea of the approach is based on two main stages: Exploration and Assessments stage and Code Restructuring stage. A case study has been carried out empirically to illustrate the proposed approach.

2. RELATED WORK

An approach that consists of two stages has been proposed for refactoring opportunities for detecting code clones in source codes [3]. The first stage is to detect the clones, which usually exist in the source code. The second stage is to distinguish between re-factorability and non-re-factorability clones [3]. However, the approach is able to check the re-factorability opportunity for clone pair, i.e., only TWU code fragments, which are detected as clones. The tool that help to remove smells, which in turn helps the developer in program boosting is described in [5]. The approach (known as Textual Analysis Technique) is to identify smells in source code. It has been instantiated for detecting long method smells and evaluated on three Java open source projects [5]. Another tool, called DECOR is designed for embodying and defining all steps necessary for the specification and detection of code smells [4]. These essential steps are developed to automate the process as much as possible. However, the tool is designed for Java legacy systems. A framework that enables the exploration, both analytically and experimentally the properties of code-improving transformations is presented in [6]. The framework includes a technique that facilitates an analytical investigation of code-improving transformations using the Gospel specifications. The approach is supported by a tool, called "Genesis", that automatically produces a transformer to implement the transformations specified in Gospel.

3. THE PROPOSED APPROACH

The proposed approach is designed for an elimination of code smells (i.e., duplicated code, long method, and large class). Figure 1 presents a schematic diagram of the stages and modules in the proposed approach. The approach is based on two main stages: (i) Exploration and Assessment Stage, and (ii) Code Restructuring Stage.

3.1 Exploration and Assessment Stage

An assessment and an in-depth analysis of code is made before the code restructuring process starts. It results an abstract code without conventions, comments and blanks, which are translated into UML class diagrams, to define software units that should be restructured. This stage helps to keep the overall picture of the software project in mind and to divide it into several sub-projects, units (or restructuring parts).

3.2 Code Restructuring Stage

Restructuring a software system means that refurbishing it internally without interfering with its external properties. It is a process of changing a software system in such a way that it does not alter the external behaviour of the code, yet to improve its internal structure. The code restructuring stage is based on a textual basis to tackle every problem individually. All expected situations (cases) are to be displayed for three types of code smells to facilitate the process of detecting these smells within the code and also to facilitate the process of choosing an appropriate solution to process each problem. In addition, at the end of this stage, the chosen solutions are applied in the source code.

4. FIRST STAGE: EXPLORATION AND ASSESSEMENT STAGE

The main goal of exploration and assessment stage is to understand and keep the overall picture of the software project that is to be restructured in mind. In addition, an in-depth analysis of software systems is made, which results in removing some unimportant material of the code, e.g., whitespace, comments and others. The analysis results in determining the software units (also known as restructuring units) constituent to software system. Once the restructuring units are determined, the source code of the restructuring units is transformed to an intermediate representation for restructuring using UML class diagrams. This transformation of the source code into an intermediate representation is often called reverse engineering process [14]. This stage is planned as in the following two main steps:

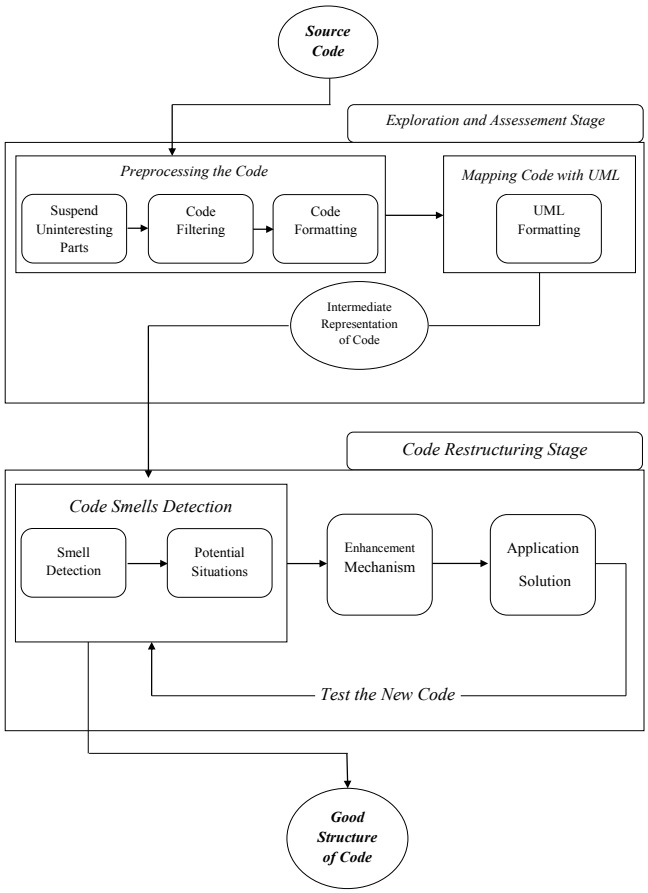

Figure 1. The Schematic Diagram of the Proposed Approach

4.1 Preprocessing the Code (Preprocessing)

Before the restructuring phase starts, units of code are transformed into an intermediate representation using UML class diagrams. The source code has to pass through several phases, in order to guarantee the following:

1. The source code is easily transformed to the intermediate representation.
2. Most importantly, the source code is partitioned and the domain of the comparison is determined.

In order to make the source code ready for transformation to a class diagram. There are three main phases that have to be completed as shown in Figure 1.

Phase I: Suspend Uninteresting Parts

All the source code uninteresting to the detection phase is filtered. In fact, these uninteresting parts is considered as one of very important parts for the software system and also the software does not work without them. However, these parts are not mainly important for our proposed approach. These uninteresting parts are not testable but are suspend at the beginning of the search of code smells in order to maintain these lines of code, which make them less subject to damage. After removing the bad smells, these uninteresting parts are returned. These uninteresting parts include:

Table 1: The source code before removing the blanks

Code Line	Original Code
1	/**
2	*
3	* **@author** sqlitetutorial.net
4	*/
5	**publicclass**SelectApp
6	{
7	/**
8	* Connect to the test.db database
9	* **@return** the Connection object
10	*/
11	****private** Connection connect()
12	**{
13	// SQLite connection string
14	**String url = **"jdbc:sqlite:C://sqlite/db/test.db";
15	**Connection conn = **null**;
16	****try**
17	**{
18	**conn = **DriverManager.*getConnection*(url);
19	**}
20	****catch** (SQLException e)
21	**{
22	System.***out***.println(e.getMessage());
23	**}
24	****return**conn;
25	**}
26	/**
27	/**
28	* **@param**args the command line arguments
29	*/
30	*/
31	**publicstaticvoid** main(String[] args)
32	{
33	SelectApp app = **new**SelectApp();
34	app.selectAll();
35	}
36	}
37	

1. Database declarations (the definitions or queries linked to the database, e.g., SQL embedded code).
2. Library directives (statements that are used for defining library in system, such as #include).
3. Some types of modifiers, which are used to define some types of variables such as the final, and constant.

The process of suspending uninteresting parts is done by putting a mark (**) at the front of the code lines that are used to link software system to the database, and at the front of the lines that are used to database queries as shown in Table 1.

Phase II: Code filtering

After the completion of the removing uninteresting parts, all code comments and blanks are removed. Tables 1 and 2 present an example for the process of removing the blanks and comments in our approach. However, the important comments that the developers cannot dispense them, as they provide very important explanations for some parts of the source code are temporarily deleted. After the approach is applied, these comments are returned.

Table 2: The source code after removing the blanks

Code Line	After filtering
1	**publicclass**SelectApp{
2	****private** Connection connect()**{
3	**String url = **"jdbc:sqlite:C://sqlite/db/test.db";
4	**Connection conn = **null**;
5	****try****{
6	**conn = **DriverManager.*getConnection*(url); **}
7	****catch** (SQLExceptione)**{
8	System.***out***.println(e.getMessage()); **}
9	****return**conn; **}
10	**publicstaticvoid** main(String[] args) {
11	SelectApp app = **new**SelectApp();
12	app.selectAll(); } }

Phase III: Code Formatting

Code formatting is the last phase in the process of pre-processing the source code. In this phase, the restructuring units of software system are determined, to be transformed into UML class diagrams. This process is performed as follows:

1. Each class of the software system, is placed in a table. i.e., represented as a restructuring unit as shown in Table 3. The table name should hold the same name of the class and given unique number.
2. The methods numbers constituent to the class are calculated, in order to help in determining the large class in the code smell in detection phase.
3. The number of code lines of each method are calculated, to help in determining the long method in the code smell.
4. If the class is related to other classes, this relationships should be identified and mentioned as follows:
   Relationship type (*Class_Name*), where: *Class_Name* is the name of the related class.

4.2 Mapping the Code with UML Class Diagram

In this stage, the restructuring units is transformed to UML class diagrams for code restructuring. This is to bridge the gap between the exploration and assessment stage and the code restructuring

stage. This transformation is performed by applying reverse engineering approach as follows:

1. Draw each class, interface or form from the table (Table 3) that represent restructuring unit using UML class diagram.
2. Identify and draw generalization (inheritance/extend) relationship between classes in the UML notations, from the relationship field in the table.
3. Identify and draw interface realization (implement) relationship between a class and an interface in the UML class diagram, from the relationship field in the table.
4. Identify and draw directed association relationship between classes from the relationship field in the table.
5. Identify and draw instantiate dependency relationship between classes from the relationship field in the table.
6. Identify and draw usage dependency relationship between classes from the relationship field in the table.
7. Draw each method in the class or interface from the table that represent restructuring unit by using the UML class diagram with method number and its name only.

Table 3: An example of identifying the restructuring units

Code Line	No of Method line	Select App	Relationship	Class Number	Method number
			None	1	2
1		**public class** SelectApp{			
2	1/1	**private** Connection connect()**{			
3	1/2	**String url = **"jdbc:sqlite:C://sqlite/db/test.db";			
4	1/3	**Connection conn = **null**;			
5	1/4	****try****{			
6	1/5	**conn = **DriverManager.*getConnection*(url);			
7	1/6	**}			
8	1/7	****catch** (SQLException)**{			
9	1/8	System.***out***.println(e.getMessage()); **}			
10	2/1	**return**conn;			
11	2/2	**public****static****void** main(String[] args) {			
12	2/3	SelectApp app = **new**SelectApp(); app.selectAll(); } }			

Table 5: Refactoring Mechanism for Long Method and Large Class

Refactoring Mechanism	**Long Method**				**Large Class**
	more than 10 lines	duplicate lines	with loops	conditional expressions	many methods
Extract Method	√	√	√		√
Parameterization		√	√		
Decompose Conditional				√	
Extract Class					√
Extract Subclass					√
Extract Interface					√

5. SECOND STAGE: CODE RESTRUCTURING

Code restructuring is performed systematically to reduce the risk of introducing bugs in the source code. The important goal here is to find the best way to achieve the new design to avoid risks. This includes discovering the order of the steps, in which an actual restructuring can be made without breaking too much code at one time. This starts by the clarification of detecting bad smells and identifying a set of cases that are associated with a given smell. Then, a list of possible refactoring for each case is proposed. Our approach is concerned to address the three major types of code smells (i.e., duplicated code, large classes and long methods) for refinement and improvement the quality of software system, taking into account to keep the external behaviour of software system. Nonetheless, each type of code smells has the unique method for detecting it in the source code, which different from other types. The following three phases describe each step of the detection process of each smell.

Phase I: Detection of Code Smells

Detection of smells inside the source code could be done in one of two ways. The first way, which is the easiest one, is by using some available tools, which are used for detection and analysis code to indicate the place of code smells. However, one of the limitations of this way is that, we have to use more than one tool consecutively in order to detect the three different types of smells. There is no one tool that could deal with the three smells. Most of the available tools detect the repetition of the code in the source code. One of the most common used tools to detect the duplication of code is: Duploc, while detection of long methods, large classes could be performed by Johnson and CCFinder tools. The second way, which is more difficult, is by using the sight (observation) to detect smells through searching units of restructuring to detect the present smells in the source code. This is done through three main ideas to detect each smell in isolation.

1. Understanding duplicated code, depending on finding the similarity in the code lines that create the source code to find the duplicated code, either in the same method or in the different classes.
2. Knowing long methods, depending on the number of method lines contained in the class. The methods that are longer than 10 lines are generally viewed as potential problem areas and can harm the readability and maintainability of the code.
3. Knowing large classes, depending on the classes that contain the methods, which have been identified as large classes. The class that has more than 10 methods is generally viewed as potential problem areas and can harm the readability and maintainability of the code.

After completion of bad smells detection inside the source code, then, the process of defining which technique of refactoring is to be used to eliminate the detected smells begins. A group of expected case for each bad smell is to be defined, in order to facilitate refactoring through suggesting a list of solutions for each smell case. This facilitates choices for the developer to get rid of smells. Tables 4 and 5 describe the cases for each bad smell and a list of proposed refactoring mechanics for each case.

Phase II: Enhancement Mechanism

Table 4: Refactoring Mechanism for Duplicated Code

Refactoring Mechanism	**Duplicated Code**						
	Same Method	Same Class	between Sibling Classes	with Superclass	with Ancestor	with First Cousin	Unrelated Classes
Extract Method	√	√	√		√	√	
Parameterization	√	√	√	√	√	√	
Insert super Call				√			
Insert Method Call		√		√			
Form Template Method		√	√	√	√	√	
Pull Up Method			√	√	√	√	
Substitute Algorithm			√				

After the detection of the smells in earlier phases, and defining the expected refactoring mechanisms for each smell, the best choices and executions are to be chosen for the best solution for restructuring. The best solutions suggested to each smell depending on the case, which appeared in the source code. This process should be conducted systematically to avoid (or reduce) entering errors in the source code. Therefore, a template has been suggested to execute the process of smell refactoring in an organized and simple way. Each refactoring is implemented according to the following template, which called a restructuring template:

1. **Bad smell**: The name of a bad smell case.
2. **Method Name**: The name of refactoring mechanism.
3. **Location**: The restructuring area(s) of the transformation.
4. **Reasons**: Probable reason(s) for performing the refactoring.
5. **Description**: A short explanation of things if its intent is not obvious.
6. **Refactoring Process**: A mechanics of the improvement–identification of basic operations and/or other refactoring and the order in which they should be applied to achieve the objectives of the process.

Phase III: Apply the solution:

In this phase, uninteresting parts (the code lines that are used to link software system to the database) are released by removing the mark (**) that are put at the front of code lines. Moreover, some important developer's comments that have been deleted from the source code are to be returned. At the end of this phase, operating program, testing its functions to ensure that restructuration is perfectly done without impacting the system behaviour.

6. CASE STUDY

The approach described in this paper has been applied, through a case study on the software of General Mills Company system. By describing our method stages, they are followed for reinforcement using Microsoft Visual Basic.Net 2010 and the techniques that are used to detect code smells. The General Mills Company system contains three classes; each of which have to be isolated from the other classes but certainly dealing with each other to perform system functionality. Besides, every class performs a set of special functions that are required. The actual implementation and further experimental details of the proposed solution to improve source code of the General Mills Company system can be found in [16].

7. CONCLUSION

This paper presents an approach for analysing, restructuring and eliminating bad smells in source code in software system, based on a reverse engineering technique. The proposed solution is a hybrid approach, as it includes two techniques, graph and text-based techniques. Graphs are used to describe all scenarios to facilitate smells detection. Texts are used in describing effective restructuring for all scenarios that represent a specific state of code smell. To validate our approach, an experiment has been conducted on General Mill's Company system, which has been analysed by using VB.Net 2010. A qualitative validation has been ratified this system before and after improvement. The results of the experiment show that the proposed approach provides a useful information, which help in detecting code smell. The use of reverse engineering has been found effective, because it supports detection of smells in a simple visual way. The restructuring definitions used in the text makes it simple to eliminate smells effectively.

8. REFERENCES

[1] Ragunath, P. and et al. (2010). Evolving A New Model (SDLC Model-2010) For Software Development Life Cycle (SDLC). In *Int. Journal of Computer Science and Network Security*, vol.10(1).

[2] Sarkar, M. and et al. (2013). Reverse Engineering: An Analysis of Static Behaviours of Object Oriented Programs by Extracting UML Class Diagram. In *Int. Journal of Advanced Com. Research*, vol. 3(12).

[3] Ashtaputre, P., Kulkarni, C. and Lonkar, Y. (2016). An Effective Approach to Find Refactoring Opportunities for Detected Code Clones. In *Int. Jour. of Innov. Research in Sci., Eng. and Tech*., vol. 5(6).

[4] Moha, N., Gueheneuc, Y. and Duchien, L. (2010). DECOR: A Method for the Specification and Detection of Code and Design Smells. In *IEEE Transactions on Soft. Eng*., vol. 36(1), pp. 20-36.

[5] Astuti, H. and et al. (2015). Software Quality Measurement and Improvement Using Refactoring and Square Metric Methods. In *Journal of Theoretical and Applied Info. Technology*, vol. 37(5).

[6] Deborah, L and et al. (1997). An Approach for Exploring Code-Improving Transformations. University of Pittsburgh: In *ACM Transactions on Prog. Lang. and Syst*., vol. 19(6), pp. 1053-1084.

[7] Sethi, D et al. (2012). Detection of code clones using Datasets. In *Int. Jour. of Advan. Res. in Comp. Sci. and Soft. Eng*., vol.7(2).

[8] Fowler, M. and et al. (1999). *Improving the Design of Existing Code*. Mass: Addison-Wesley.

[9] Navita. (2017). A Study on Software Development Life Cycle & its Model. In *Int. Journal of Eng. Research in Computer Science and Engineering*. Vol 4(9), pp. 2394-2320..

[10] Astuti, H. et al. (2015). Software Quality Measurement and Improvement Using Refactoring and Square Metric Methods. In *Journal of Theoretical and Applied Info. Technology*, vol. 37(5).

[11] Abdelaziz, T. M., Maatuk A. M. and Rajab F. (2016). An Approach to Improvement the Usability in Software Products. In *Int. Journal of Software Engineering & Applications (IJSEA),* vol.7(2), DOI : 10.5121/ijsea.2016.720211

[12] Zibran, M. and Roy, C. (2013). Conflict-aware optimal scheduling of prioritised code clone refactoring. In *11th IEEE Int. Working: Conference on Source Code Analysis and Manipulation.*

[13] Yamashita, A. (2012). *Assessing the Capability of Code Smells to Support Software Maintainability Assessments: Empirical Inquiry and Methodological Approach*. Department of Informatics Faculty of Mathematics and Natural Sciences University of Oslo.

[14] Maatuk, A. M., Ali A. and Rossiter, N. (2011). Re-Engineering Relational Database: The Way Forward. In *Proc. of the 2nd Int. Conf. on Intelligent Semantic Web-Services and Applications, Jordan, ACM*, pp. 18.

[15] Elmarzaki, H. and Abdelaziz, T. M. (2018). Increasing the Architectures Design Quality for MAS: An Approach to Minimize the Effects of Complexity. In *7th Int. Conf. on Soft. Eng. and Applications.*

[16] Elgathafi, H. A. (2018). *An Enhancement Approach of Software System-Using Reverse Engineering and Restructuring Concept to Improve the Quality of Software Code*. Benghazi University, Faculty of Information Technology.

[17] Abdelaziz, T. M., Zada, Y. and Hagal, M. A. (2014). A Structural Approach to Improve Software Design Reusability. In *3rd Int. Conf. on Info. Tech. Convergence and Services*, pp. 163-171.

[18] EL-firjani N. F., Elberkawi E. K. and Maatuk, A. M. (2017). A Method For Website Usability Evaluation: A Comparative Analysis. In *Int. Jour. of Web & Semantic Technology (IJWesT)*, AIRCC Pub. Co, vol.8(3), pp. 1-11.

[19] Shadi A. Aljawarneh, Muneer Bani Yassein, and We'am Adel Talafha. 2018. A multithreaded programming approach for multimedia big data: encryption system. Multimedia Tools Appl. 77, 9 (May 2018), 10997-11016. DOI: https://doi.org/10.1007/s11042-017-4873-9

[20] Vangipuram Radhakrishna, Shadi A. Aljawarneh, Puligadda Veereswara Kumar, and Kim-Kwang Raymond Choo. 2018. A novel fuzzy gaussian-based dissimilarity measure for discovering similarity temporal association patterns. Soft Comput. 22, 6 (March 2018), 1903-1919. DOI: https://doi.org/10.1007/s00500-016-2445-y

[21] Shadi A. Aljawarneh, Muneer Bani Yassein, and We'am Adel Talafha. 2017. A resource-efficient encryption algorithm for multimedia big data. Multimedia Tools Appl. 76, 21 (November 2017), 22703-22724. DOI: https://doi.org/10.1007/s11042-016-4333-y

[22] Shadi A. Aljawarneh, Ali Alawneh, and Reem Jaradat. 2017. Cloud security engineering. Future Gener. Comput. Syst. 74, C (September 2017), 385-392. DOI: https://doi.org/10.1016/j.future.2016.10.005

[23] Shadi A. Aljawarneh, Radhakrishna Vangipuram, Veereswara Kumar Puligadda, and Janaki Vinjamuri. 2017. G-SPAMINE. Future Gener. Comput. Syst. 74, C (September 2017), 430-443. DOI: https://doi.org/10.1016/j.future.2017.01.013

[24] Muneer Bani Yassein, Shadi A. Aljawarneh, and Esraa Masadeh. 2017. A new elastic trickle timer algorithm for Internet of Things. J. Netw. Comput. Appl. 89, C (July 2017), 38-47. DOI: https://doi.org/10.1016/j.jnca.2017.01.024

[25] Shadi A. Aljawarneh, Mohammed R. Elkobaisi, and Abdelsalam M. Maatuk. 2017. A new agent approach for recognizing research trends in wearable systems. Comput. Electr. Eng. 61, C (July 2017), 275-286. DOI: https://doi.org/10.1016/j.compeleceng.2016.12.003

[26] Muneer O. Bani Yassein and Shadi A. Aljawarneh. 2016. A Conceptual Security Framework for Cloud Computing Issues. Int. J. Intell. Inf. Technol.12, 2 (April 2016), 12-24. DOI=http://dx.doi.org/10.4018/IJIIT.2016040102