Software Refactoring Approaches: A Survey

Ismail M. Keshta
Department of Computer Engineering
King Fahd University of Petroleum and Minerals
Dhahran, Saudi Arabia

***Abstract*—The objective of software refactoring is to improve the software product's quality by improving its performance and understandability. There are also different quality attributes that software refactoring can improve. This study gives a wide overview of five primary approaches to software refactoring. These are two clustering approaches at class level and two at package level, as well as one graph transformational approach at class level. The research also compares the approaches using several evaluation criteria.**

***Keywords*—Software refactoring; refactoring tool; machine learning; hierarchical clustering; graph transformations**

I. Introduction

Due to its properties in a real-world environment, as well as changes to requirements, software needs to evolve, leading to both improvements and alterations. Therefore, the software becomes increasingly complicated and changes from its original design in some way. Adding features generally deteriorates the product's design, and the program therefore becomes more complex as it evolves. Consequently, the product's quality decreases [1][2][3]. This means maintaining the code is a vital task, as it decreases the software's complexity. The maintenance of software is considered one of software development's major parts.

A vital kind of maintenance is a process called refactoring. This is defined as a method for restructuring a current software system or body of code. This refactoring is carried out in the system/code of the internal structure to carry out improvements without altering external behaviour. As a result, software projects using the refactoring process discover reductions in the code base's complexity[2]-[4].

Crucially, there is no single definition of software refactoring that is universally accepted. It is merely the process of altering the internal structure of a software system without changing its external behavior [5][6]. In doing this, the refactored code can have optimised object-oriented features, including encapsulation, polymorphism, and inheritance, that can improve the quality of the code's maintainability, reusability, and modifiability. Software refactoring's key purpose is, in most cases, to improve the quality of a program by decreasing any shortcomings in quality, such as code smells, anti-patterns, and anomalies [7][8][9].

Therefore, the most significant motivation for refactoring is to increase the software product's quality. The major quality aspects of a software product are its understandability, extensibility, and maintainability, which can be developed by software refactoring without changing the software product's functionality [10].

It is vital to point out that the refactoring process consumes time. It also reduces the internal complexity of software because it requires an effort to first identify where to carry out the process in a given system/code and then a decision about what refactoring approach is the best to apply [11]. Furthermore, a common concern is the effect the process has on the program's performance, as the change may make it run more slowly [12]. In addition, it means the software is more capable of performance tuning [13].

Over the past fifteen years, researchers have contributed a great deal of knowledge and many concepts to the field of software refactoring, which cover various angles and different phases of software development activities, such as software design, requirement analysis, integration ,implementation, maintenance, and testing [14]. The term *software refactoring* is very much associated with, and used regularly in, coding activity (generally known as code refactoring). It is therefore necessary to gain the right skills, knowledge, tools, and techniques to benefit fully from software refactoring [13], [14].

A wide range of techniques and formalisms are proposed in software refactoring to deal with restructuring and refactoring, such as software metrics, graph transformations, and assertions. This refactoring can be carried out either manually or by using various supporting tools. Many of the available tools can automate the different aspects of refactoring [15].

A number of recent papers on the three fundamental software refactoring approaches, including the clustering approach at class level, the clustering approach at package level, and the graph transformations approach, will be presented in this short survey work. In addition, we will provide a comparison between the represented software refactoring approaches based on various existing evaluation criteria.

The structure of this paper is as follows: Section 2 will provide a literature review of the different software refactoring approaches and classify them according to their refactoring levels; Section 3 will list the evaluation characteristics; Section 4 will provide discussions and a comparison between the different approaches; and finally, Section 5 will explore a conclusion and future directions.

II. Software Refactoring Approaches

The software refactoring approaches presented here include a clustering approach at the class level, a clustering approach at the package level, and a graph transformation approach. We will both list and summarise them throughout this section. We

will also describe the mechanisms and features for each of the approaches.

A. Software Refactoring at the Class Level using Clustering Techniques

There are two important concepts at the software design stage, which are coupling and cohesion. The first of these, coupling, represents the various interdependencies among the software modules. However, a model's relative functional strength is indicated by its cohesion. Therefore, a software design should ideally possess low coupling and high cohesion.

The authors' objective in [16] was to use clustering techniques of different kinds that help to both maximise cohesion and minimise coupling. This means that software designers can easily refactor the software code at the class level. To maximise cohesion and minimise the coupling process, it is necessary to move some of these methods from one class in a system to another. The authors, therefore, used two approaches.

The first is refactoring at the class level. This is achieved by clustering a fixed number of different classes, which means there is a movement of the method from one class to another. The total number of classes, however, remains unchanged before and after refactoring. There are potential changes to the number of classes in the second approach, therefore, the number of classes in a system can be different before and after refactoring.

Clustering is a technique generally used to group all similar data sets into the same cluster. Other dissimilar entities, however, are grouped into different clusters. The greatest advantage of such a technique is that it can help to identify items that are unstructured.

A method to identify unstructured software code at the class level was proposed by the authors. In the method, a total of three different clustering techniques were utilised for identification. These three are the single linkage algorithm (SLINK), as well as the complete-linkage algorithm (CLINK) and the weighted pair group method, which uses arithmetic averages (WPGMA). An additional algorithm used is the adaptive k-nearest neighbour (A-KNN), and a comparison between the A-KNN technique and the other three clustering techniques (SLINK, CLINK, and WPGMA) was performed by the authors. The results of this comparison show that software structuring at the class level that uses A-KNN has a competitive performance with lower computational complexity when compared to the other clustering techniques, SLINK, CLINK, and WPGMA.

1) The authors' clustering process

All the entities, as well as the attributes, must be specified in the clustering process. Entities are those items needing to be grouped. It is considered that the methods are those entities in software refactoring taking place at the class level process. The authors decided that the methods are entities, as the main computational elements of the classes are done in the methods. The entities are the methods necessary to put them into clusters.

To put clustering entities into clusters, all the features and attributes of these entities must be extracted. These features and attributes are utilised to measure the relationship between two entities and how closely they are related. The entities and the methods in the case are all similar if they share more common features and attributes. The authors consider the class data members as features for the entities. The number of times the method has accessed the data members is known as the feature value.

The authors used an entity-feature matrix to represent the relationship. The rows found in this matrix represent the methods and columns that represent the data members. In this matrix, there are three types of matches utilised for any two of the entities. The first type is n-0/0-n.This means that the two entities have no-match, and so are dissimilar. The second type is n-m. This means that the two entities share at least one feature. The third type is a 0-0 match. Here, the two entities have no feature that is accessed by them. Generally, two methods will be in the same class if they are found to share many of the features. This means they are closely related to each other and this process will make the code more cohesive.

To measure the similarity between the two entities/ methods, the authors used a coefficient known as a resemblance coefficient. This is used to determine the similarity of the matrix's values. The formula for the resemblance coefficient is given by:

Coeff = similarity actor/ (similarity actor/dissimilarity actor)

The authors used three clustering techniques for the data clustering, which are SLINK, CLINK, and WPGMA. These are examples of agglomerative hierarchical clustering methods. The agglomerative process begins with the entities/methods taken as individual clusters. At each step, the closest pair of clusters merges until only one of the clusters is left.

The difference between the three techniques is in the way the distance between the clusters is computed. The distance between the nearest pair of elements is the distance between the clusters in SLINK. The similarity between the two clusters in CLINK is the similarity between their most dissimilar members. The average of the various distances between all the pairs of elements is the distance between the clusters in WPGMA.

The authors, besides using these clustering techniques, used the A-KNN algorithm. The first step of A-KNN is the same as in the previous techniques, as it considers each method as a cluster. As an additional step, the algorithm utilises a labelling approach. Thus, each method has a unique label that can be used as an identifier to the cluster. There are different values of K that can be utilised in the A-KNN algorithms. The major advantage of the A-KNN algorithm is that it reduces the number of computations when compared with the previous algorithms.

2) Two refactoring approaches at the class level used by the authors

The authors used two approaches for refactoring at the class level.

a) Refactoring at the class level that uses clustering with a fixed numberof classes

The total number of classes remains unchanged in the system with this approach, both before and after refactoring. The entities are those methods that need to be put in a cluster and the classes are the clusters. Therefore, the number of clusters is the number of classes needed in the system. A method is assigned to a class based on the similarity value. The value is calculated by the number of instances utilised by the methods. So, if method X uses a total of three instances from class A and two instances from class B, method is assigned to the cluster that represents class A.

As an overview of this similarity matrix, each of the methods will be in a row and each of the classes will be in a column. The similarity matrix value between the method and the corresponding class is the total number of instances of this class used by this method.

b) Refactoring at class level by utilising clustering with an adaptive number of classes

There will be potential changes to the number of classes in this approach. The total number of classes in a system may be different both before and after refactoring, so there are no restrictions on the number of classes in this approach.

3) Results

The authors conducted experiments to find out how effective their proposed approaches proved to be. Both approaches were software refactoring at the class level. When a comparison of these two approaches was made, the authors found that the first approach (Approach1) gave the same distribution for the methods inside the classes as was the case in the original source code. This method can, therefore, be used as an automatic method to check the consistency between the distribution for the methods inside the classes in the original source code and the distribution that was produced by Approach 1.

They found that the second approach (Approach 2) suggested there was a different distribution for the methods inside the classes which decreased the value of the lack of cohesion in methods (LCOM) metric in the system. This approach, therefore, increases the original code's quality.

Thus, the first approach is generally both easier and more simple than the second one. Furthermore, the first approach does not require any extra effort or computation. The second approach, however, improves the original code's quality and provides better refactoring suggestions than the first one.

B. Software Refactoring at the Package Level using Clustering Techniques

The authors' objective in [17] was to carry out an investigation of software refactoring at the package level, which was done by utilising clustering techniques. This research helps to identify any ill-structured packages, and their approach helps to create a balance between intra-package cohesion and inter-package coupling. Thus, software designers who use the authors' approach can refactor their software easily at package level. In the same way as the previous paper [16], a comparison is made of the behaviour of four differing techniques, which are SLINK, CLINK, WPGMA, and A-KNN, but this time it is to identify any ill-structure at the package level instead of the class level.

1) Clustering process

The same clustering process was used as in [16], but with a different context. Classes are chosen as entities for software refactoring at the package level. This means that the entities are the classes that need to be put into clusters. At the class level, however, the methods are chosen.

The attributes of the entities must be extracted in order to put these entities into clusters, as stated in [16]. The relationship between the two entities is indicated by their features. So, if the two entities share features that are more common, they will be similar. The authors utilised the methods as attributes for the entities/classes for refactoring at the package level, however, they used class data members as attributes of refactoring at the class level. The number of times the class accessed the method that was represented by an attribute was used for the features value, while the number of times the method accessed data members was the feature value for refactoring at class level.

With the package, the number of times that class was used as a class attribute inside it indicates the similarity between a package and a class. In other words, at class level, the similarity that exists between a method and a class is the number of times that class is used by the method.

The authors used the entity-feature matrix to represent the relationship between the entities and corresponding features. The rows of this matrix represent the classes in the packages and the columns represent the methods, but for refactoring at class level, the rows in this matrix represent the methods and the columns represent data members. The authors used the same coefficient as in [16] to measure the similarity between two classes, which is a resemblance coefficient.

2) Two approaches for refactoring at the package level used by the authors

The authors used two approaches for refactoring at the package level:

a) Clustering with a fixed number of packages

There is movement of a class from one package to another in this approach, but the number of packages is unchanged. Therefore, the total number of packages is the same in the system before and after refactoring.

b) Clustering with a variable number of packages

There is movement of classes between the packages in this approach, with possible changes to the number of packages. Therefore, the total number of packages in a system may be different before and after refactoring. There is no restriction on the number of packages in a system in this approach. Therefore, new packages can be created and existing packages deleted after refactoring by this approach. More packages are necessary if there is a low similarity between the classes, and fewer are necessary if there is a great deal of similarity between the classes.

3) Results

When the A-KNN algorithm is used in the first approach, it increases the number of connections inside the package, and therefore improves cohesion as a result. This algorithm also decreases the number of connections to other packages, which means that the amount of coupling between the packages is reduced. The authors' conclusion was that A-KNN improves software quality by both minimising package coupling and maximising package cohesion.

In the second approach, all three clustering techniques—SLINK, CLINK, and WPGMA—suggest the same solution. Also, the number of connections both inside and outside the packages is not changed by these three different techniques. The number remains the same both before and after the refactoring process. In other words, the A-KNN technique changes this number to increase package cohesion and decrease package coupling. Thus, software quality is improved by software refactoring at the package level by using A-KNN clustering with a variable number of packages. After carrying out a deep analysis of the results obtained, the authors concluded that A-KNN shows a competitive performance with lower computational complexity when compared to the three clustering techniques, SLINK, CLINK, and WPGMA.

C. Graph Transformation Approach to Refactoring

The paper primarily considers the methods and concepts from the graph transformation theory to locate the dependencies between the various refactoring steps. The graphs are used as abstract representations for most of the model. As is evident, a graph contains a set of vertices V, as well as a set of edges E. It is important to highlight here that an edge in E has both a source and a target in V. Thus, the programs are represented as graphs to make them more understandable and refactorings correspond to the graph transformations' production rules. The authors point to the use of graph transformations as a way of reasoning about the dependence that exists between refactorings. Moreover, the graph transformation approach aids in the sequential dependencies analysis between refactorings [18], [19].

To improve the design, we need to discover the correct sequence of refactorings from a given set of refactorings. To do this, the construction graph must be set by representing the set of proposed refactorings as nodes in the graph (G). The edges of the graph represent the various dependencies or conflicts that exist between the different refactorings in the set of refactorings that is proposed. After this construction graph has been completed, there is a highly formal way to represent all the potential interactions between the refactorings. This constructed graph will help by giving a clear summary of the refactorings proposed. It also enables us to find the dependencies between them, as well as their form and critical pair analysis technique.

The graph transformation rules, which are p: L → R, are used to detect the dependency between the instances of type graphs. In these rules, L is the left-hand of the rule and R is the right-hand. L represents the preconditions of the rule in the transformation rule, while R describes the post conditions. There is an intersection between L and R that must be very clearly defined. The preconditions and post conditions have to be formulated, as was previously mentioned, and then checked both before and after the refactoring process is applied [20], [21]. The steps taken by the authors were:

- To represent the system as a graph.
- To represent the model refactoring as a graph transformation.
- To represent the individual refactoring steps as the nodes of a graph. "The edges represent the dependencies or the conflicts between the refactorings in the proposed set of refactorings."
- To search for an optimal path that represents the best possible sequence of the refactoring steps. "Searching problem for optimal sequence" by using metaheuristic algorithms.

The authors of this paper [20] have focused on working out how the refactoring process can be formulated as a graph. Thus, they have proposed a local formulation of this refactoring that is based on graph transformation. The authors used the graphs to represent the software architectures at the class level in this research work. For the formalisations of the refactoring operations, the graph transformation was used. Their primary goal was to provide an automated process to select refactoring sequences that are appropriate and to formulate this as an optimisation problem by utilising the ant colony optimisation (ACO). This is a paradigm for the design of metaheuristic algorithms for the various combinatorial optimisation problems.

III. Evaluation Characteristics

In section 2 we described the software refactoring approaches. In this section, we will list the evaluation characteristics (see TABLE I) that we used in the comparisons between the approaches.

A. Objective

This characteristic will determine the objective and aim of the approach. For example, the authors propose approach X, as they want to address maintainability, and someone else would like to address another performance issue.

B. Level of Refactoring

This will determine what the aim of the software refactoring approach is. It will, therefore, determine whether this proposed approach will address the appropriate refactoring level. Here, the approach that is proposed should determine the appropriate level of refactoring to apply.

C. Tool-Supported "Supportability"

The best way to refactor software code is, in most cases, manual refactoring because altering these codes requires human consideration. The refactoring tools can improve the quality of software, aiding in carrying out automated changes in the software code. The tool-supported characteristic will, therefore, indicate whether the proposed approach possesses a tool support. If so, the main characteristic of this tool is highlighted, such as its usability or efficiency.

TABLE I. DEFINITION OF THE EVALUATION CRITERIA

Characteristic	Brief description
Objective	• Determines both the aim and the objective of the approach • What is the author's main objective in using this approach? • What do they aim to reach? • What performance issues do they want to achieve with this approach?
Level of refactoring	• Determines what the proposed approach is addressing, and at which refactoring level • At which level is it appropriate to apply the refactoring?
Supportability	• Indicates whether the proposed approach possesses tool support • Is this is a tool-supported approach?
Constraint	• Describes what occurs to the software artifacts while the software refactoring processes are taking place • How many software artifacts are there before and after the refactoring process?
Underlying concepts	• Which algorithm type is used by the approach?
Complexity	• Determines the complexity of an approach; a higher complexity approach uses both a complex formula and an algorithm • How complex are the refactoring steps? • What is the complexity of the algorithms used?
Validity	• Determines whether the proposed approach is a valid one or not; explains whether the approach can be applied to the real system • Is this a valid approach? • Can the approach be applied to the real system?

D. Constraint

A software artifact is an element of a software project, which includes images, class, documentation, modules and package. This evaluation characteristic, therefore, highlights the total number of artifacts there are before as well as after refactoring.

E. Underlying Concepts

This characteristic indicates the algorithm type that is used in the proposed software refactoring approach.

F. Complexity

If the software refactoring approach uses a complex formula and algorithm, the approach is said to have a higher complexity. This means the complexity measures how complex the refactoring steps are.

G. Validity

The characteristic of validity will determine whether the proposed approach is a valid one or not. Therefore, it will explain whether this approach can be applied to the real system.

IV. DISCUSSION

An important criterion for evaluating the different approaches is objectivity. This characteristic determines both the aim and objective of the approach. The main objective of the clustering approach is similar at both the class and package levels. The aim of clustering at the class level is to identify the unstructured software code and then structure it in an improved way that can make it much more understandable. This would, as a result, give it high maintainability. The aim of clustering at the package level is to identify ill-structured packages and find a balance between package cohesion, on the one hand, and package coupling, on the other. The primary objective of the graph transformations approach is to improve the system's performance (scalability).

When it comes to the level of refactoring, the first method proposed in the first paper clearly addresses the software refactoring done at class level. The authors [16] used two approaches for refactoring, and both are at class level. The authors [17] also used two approaches for the second method of software refactoring, and both were at the package level, which helped to identify the ill-structured packages. Therefore, it is at the package level that the graph transformations approach will most likely address the architectures at class level.

Generally, the clustering approaches do not have a fully tooled, supported "supportability" at both the class and package levels. They are using some tools in the intermediate steps, but there is no tool to do the whole of the refactoring process completely. The graph transformations approach uses the GT tool.

In terms of the constraints, we will highlight what happened to a number of the software artifacts while the software refactoring processes were taking place. The total number of classes remained the same before and after refactoring when it came to refactoring at class level. In this instance, clustering was used with a fixed number of classes, but there is no restriction on the number of classes when refactoring is done at class level, where clustering is used with an adaptive number of classes. At package level, where there is clustering with a fixed number of packages, the total number of packages remained the same both before and after the refactoring process. There is, however, no restriction on the number of packages with a variable number of packages. It is most likely that the graph transformations approach will increase the number of classes after the refactoring process has been completed.

For the underlying concepts, the authors used the clustering algorithms, SLINK, CLINK, WPGMA, and A-KNN, for the task of clustering and to compare the behaviour of four different algorithms. The authors concluded that, in both cases, A-KNN showed a competitive performance with a computational complexity that was lower when compared with SLINK, CLINK, and WPGMA. The graph transformations approach utilises the theory of graph transformation as one of its main concepts.

The characteristic of complexity indicates what algorithm type is used by the software refactoring approach that is proposed. The first and second methods, with their differing approaches, are using different clustering algorithms. The results obtained reveal that the software structuring, by using A-KNN, shows a competitive performance with a lower computational complexity when compared with the other clustering algorithm. The question is still "What is the best

value of k to choose?" The findings indicate that the best results were achieved with k=3, however, this may not always be the case. It is difficult to map the class in the graph transformations approach and to define the precondition and post condition.

We can see that in the validity aspects, the clustering approach may not be valid. This is because the approach for the test at class level was carried out on an open source system. That project is called CSGestionnaire. Even at the package level, the test was done on the Trama project. Generally, we can say that these approaches are still valid for the small system, but this is not the case when it comes to the real complex system.

V. CONCLUSIONS

The technique of software refactoring can transform the different types of software artifacts to enhance the internal structure of the software without affecting its external behaviour. Refactoring is usually applied to improve the quality of the software after several features have been added. Researchers in this field have studied the various angles of refactoring and developed the right levels of evidence, skill, and knowledge. They have also published their findings in journals and conference papers to make them accessible to everyone.

This study's main purpose was to highlight some of the main challenges faced in software refactoring. Furthermore, the five refactoring approaches were discussed. These were two clustering approaches at class level and two at package level, as well as the graph transformations approach at class level. The evaluation characteristics that were used to compare the approaches were also described. Finally, researchers have contributed a great deal to the software refactoring field over the last 15 years, but there are many unresolved issues that will need to be addressed in the future. The gaps that have been identified and the significant contributions that have been made can guide researchers regarding the best areas on which to focus. This can save time and effort as well as resources, and reduce the need to reinvent the wheel.

REFERENCES

[1] MURPHY-HILL, E., AND BLACK, A. 2008. Refactoring tools: Fitness for purpose. Software. 25, 5, 38-44.

[2] Fowler, M., & Beck, K. (1999). Refactoring: improving the design of existing code. Addison-Wesley Professional.

[3] Czibula, I. G., & Serban, G. (2007). Hierarchical clustering for software systems restructuring. INFOCOMP Journal of Computer Science, 6(4), 43-51.

[4] Qasim, S. Z., & Ismail, M. A. (2017). Research problems in Search-Based Software Engineering for many-objective optimization. In *Innovations in Electrical Engineering and Computational Technologies (ICIEECT), 2017 International Conference on* (pp. 1-6). IEEE.

[5] Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999). Refactoring: Improving the design of existing programs.

[6] Chávez, A., Ferreira, I., Fernandes, E., Cedrim, D. and Garcia, A., (2017). How does refactoring affect internal quality attributes?: A multi-project study. In *Proceedings of the 31st Brazilian Symposium on Software Engineering* (pp. 74-83). ACM.

[7] Shatnawi, R., & Li, W. (2011). An empirical assessment of refactoring impact on software quality using a hierarchical quality model. International Journal of Software Engineering and Its Applications, 5(4), 127-149.

[8] Ó Cinnéide, M., Yamashita, A., & Counsell, S. (2016, September). Measuring refactoring benefits: a survey of the evidence. In Proceedings of the 1st International Workshop on Software Refactoring (pp. 9-12). ACM. Chicago

[9] Vimaladevi, M., & Zayaraz, G. (2017). Stability Aware Software Refactoring Using Hybrid Search Based Techniques. In *Technical Advancements in Computers and Communications (ICTACC), 2017 International Conference on*(pp. 32-35). IEEE.

[10] Arora, M., Sarangdevot, S. S., Rathore, V. S., Deegwal, J., & Arora, S. (2011). Refactoring, way for software maintenance. International Journal of Computer Science Issues, 8(2), 565-570.

[11] Massoni, T., Gheyi, R., & Borba, P. (2005). Formal refactoring for UML class diagrams. In Proceedings of the 19th Brazilian Symposium on Software Engineering (pp. 152-167).

[12] https://sourcemaking.com/refactoring

[13] Abebe, M., & Yoo, C. J. (2014). Trends, opportunities and challenges of software refactoring: A systematic literature review. International Journal of Software Engineering & Its Applications, 8.

[14] Arnold, R. S. (1986). An introduction to software restructuring (pp. 1-11). IEEE Computer Society Press, Washington, DC.

[15] Pérez, J. (2006, July). Overview of Refactoring Discovering Problem. In Doctoral Symposium, 20th edition of the European Conference on Object-Oriented Programming (ECOOP 2006).

[16] Alkhalid, A., Alshayeb, M., & Mahmoud, S. A. (2011). Software refactoring at the class level using clustering techniques. Journal of Research and Practice in Information Technology, 43(4), 285.

[17] Alkhalid, A., Alshayeb, M., & Mahmoud, S. A. (2011). Software refactoring at the package level using clustering techniques. IET software, 5(3), 274-286.

[18] Van Eetvelde, N., & Janssens, D. (2003). A hierarchical program representation for refactoring. Electronic Notes in Theoretical Computer Science, 82(7), 91-104.

[19] Campbell, D., & Miller, M. (2008). Designing refactoring tools for developers. In Proceedings of the 2nd Workshop on Refactoring Tools (p. 9). ACM.

[20] Qayum, F., & Heckel, R. (2009). Local search-based refactoring as graph transformation. In Search Based Software Engineering, 2009 1st International Symposium on (pp. 43-46). IEEE.

[21] Katić, M., & Fertalj, K. (2009). Challenges and discussion of software redesign. In Proceedings of the 4th International Conference on Information Technology.