Approaches for Refactoring to Frameworks

Mohammad Alshayeb and Faisal Banaeamah

Information and Computer Science Department
King Fahd University of Petroleum and Minerals
Dhahran 31261, Saudi Arabia
{alshayeb, fmb}@kfupm.edu.sa

Abstract

Software refactoring is the process of improving the internal structure of the software while not affecting its external behavior. Refactoring to framework is a software refactoring process that is applied to an existing software application to produce reusable domain classes while improving their quality. These produced classes form the software architecture and can be reused in the development of other applications. In this paper, we propose two approaches for refactoring to framework; the quality attribute based refactoring to framework (QARtF) and the level based refactoring to framework (LRtF). We empirically validate these two proposed approaches. Results show that these two approaches can produce high quality domain classes that can be used to form software frameworks.

Keyword: Refactoring to framework; software architecture; quality improvement.

I. Introduction

Application frameworks are semi-complete applications that can be reused to produce custom applications [1]. The application frameworks tend to improve software quality through building reusable components. These reusable components can be used further by other applications. However, the application frameworks improve software quality by localizing the impact of design and implementation changes. Thus, the effort required for understanding and maintaining existing software is reduced [1]. Examples of frameworks include: Microsoft Foundation Classes (MFC),

Distributed Component Object Model (DCOM) and Common Object Requesting Broker Architecture (CORBA) [1].

Software refactoring is restructuring internal structure of an existing software application without affecting its external behaviors. The software refactoring can also be used to improve the software quality [2].

Refactoring to framework is a software refactoring process which is applied to an existing software application to produce an application framework (domain classes). In refactoring to framework, the internal structure of the software application is changed to improve certain software quality goals that are related to framework. The produced set of domain classes (framework) can be reused in future to implement other custom applications. Refactoring to framework process suggests a sequence of refactoring methods to be applied on any software application in order to produce a set of domain class that can be used as a framework.

It is expected that the refactoring to framework improves the software quality of software applications. This is because the application frameworks enhance software quality via building reusable components [1]. Therefore, the software quality of the software applications is improved when reusing the application frameworks.

There are a number of methods exist that help designers create frameworks [3, 4], however, none of these methods provide guidelines or steps to refactor existing software applications to frameworks. Therefore, the objective of this paper is to propose a process for the refactoring to framework which consists of a set of refactoring methods that can be applied in sequence to produce a framework. We propose two approaches for refactoring to framework, these are: quality attribute based refactoring to framework (QARtF) and the level based refactoring to framework (LRtF). These refactoring to

framework processes are empirically validated using two different software applications from different domains with different sizes.

II. Literature Review

Software refactoring has been investigated extensively. In this section we review the work that was done in the areas of improving quality using refactoring, refactoring processes, and automating refactoring using tools.

Tiarks discussed refactoring a system due to specific quality requirements, such as the maintainability [5]. He considered relationships and dependencies between quality goals of software. The result was a framework to visualize quality requirements and their dependencies using software metrics to evaluate and measure the refactoring quality [5]. Tahvildari showed a framework of re-engineering of object-oriented systems to improve software quality [6]. The framework considered specific design and quality requirements namely, performance and maintainability. The evaluation procedure upon each transformation step was done using software metrics.

There are many refactoring processes which are followed to perform refactoring on applications. Some processes target implementation phase while others target the design phase [5-7] .Tahvildari et al. proposed a refactoring process for object-oriented legacy systems to improve the fulfillment of the non-functional requirements, such as reusability, performance and maintainability [8]. Geppert et al. applied refactoring on a part of a large legacy business communication product [9]. They proposed a number of strategies and effects of the refactoring effort for changeability. Kolb et al. described several-model refactoring processes of systematic refactoring of an existing software component for reuse in a product line by improving the design and implementation for reusability and maintainability [10]. As advantage of their work, maintainability, reusability and hence

suitability of a legacy product line were increased [10]. However, their work is limited to quality needed for legacy system, not for any type of application. Refactoring with contract [11], or refactoring by contract (RbC) [12], is another process of refactoring. It is a refactoring technique to verify refactoring based on contracts [11, 12]. Bryton and Abreu proposed modularity-oriented refactoring process (MORe) and developed a MORe Eclipse plug-in tool [13]. The modularity-oriented refactoring is a cross-paradigm and language independent refactoring process, based on modularity metrics [13]. S. Shrivastava and V. Shrivastava presented a metrics based refactoring process to improve software quality [14]. Tsantalis and Chatzigeorgiou proposed a methodology to automatically identify Extract Method refactoring opportunities [15]. The refactoring methodology adheres on preserving behavior of a program after refactoring, containing all computations of variables declarations in extracted code and duplicating extracted code in original method [15].

Refactoring of software architecture is the first step in maintaining system quality during evolution [16]. This is because software architecture is the highest level of design and implementation phases and the first step of matching requirements. Ivkovic and Kontogiannis introduced a framework of software architecture refactoring using model transformations and quality improvement semantic annotations [16]. Grunske and Neumann proposed a refactoring process focused on architecture specifications such as safety, reliability, maintainability, availability and temporal correctness [17]. Each component of the application refactored should be annotated with an evaluation model such as Component Fault Trees (CFT).

Refactoring tools automate refactorings rather than refactoring manually with an editor [18-20]. Many development environments of different programming languages include refactoring tool [18, 21], for example, Eclipse, Microsoft Visual Studio, Xcode and Squeak [18]. The main advantages of the refactoring tools are to make the refactoring process less mistakable and faster [18], (i.e. quick and correct [22]).

III. Application Frameworks Quality Attributes

The application framework quality attributes are set of external quality attributes which characterize application frameworks. The application frameworks quality attributes are reusability, modularity, extensibility, flexibility, maintainability [1] and usability [3].

3.1 Reusability

Reusability is the ability of using software components or an existing source code in the development of another software application. It improves quality of the software application [23]. The reusability of software is one of the most essential benefits and characteristics of the application frameworks [1]. Reusability of software can be addressed by the generalization of software via generalizing common features of software [24]. The reusable software should extend and reuse existing software components [24, 25] and capture components constraints and design decisions [24]. The reusability of software also relies on abstract software architecture [25]. The reusability of software is enhanced by creating documentation and comments [24]. However, the reusability is affected by some other quality attributes such as complexity, testability and modularity [23, 26].

3.2 Modularity

Modularity is the ability of making software more consisting of separate independence parts or components, called modules [23]. The modularity of software is a primary feature of the application frameworks [1]. Modularity is affected by cohesion and coupling [23].

3.3 Extensibility

Extensibility is the ability to extend a system, with minimum level of implementations and minimum impact on existing functions. It is a system design principle that takes into consideration future

growth [27]. Software extensibility is one of the primary benefits and features of the application frameworks [1]. The extensibility of software is enhanced by providing syntactic definitions for all extensible software components. It is as well improved by providing precise semantic definitions and documentations [27].

3.4 Maintainability

Maintenance is the process of implementing corrective, adaptive or perfective software changes [28]. Maintainability is defined as the ability of a software application to be indicative of amount of effort necessary to perform maintenance changes [28]. The software maintainability can be estimated by the average maintenance effort and the complexity of software [28, 29]. The maintainability of software can be improved using hierarchical multidimensional design methodology. The hierarchical multidimensional design methodology consists of decomposing a software application into hierarchical levels from different dimensions; control, information and typography structures [28].

3.5 Usability

Usability can be defined as the ease of use of a software application or product [30, 31]. It also means the level of ease to understand, learn, and use a software application [31]. The usability of software can be measured by different criteria such as the understandability of the software application [32]. The usability of software is affected by the functionality, consistency and self-explanatory of software components and messages [30]. The source code of software becomes usable through aggregating and combining data, long-running statements and multi-step commands [24].

3.6 Flexibility

Flexibility is the ease with which a software application or component can be modified for use in applications or environments other than those for which it was specifically designed for [33]. The flexibility of software is affected by the evolution cost and evolution complexity [33].

IV. Refactoring Improvements on Software Quality

In the previous section, we identified the application framework quality attributes. In this section, we compose a list of refactoring methods that positively contribute to the quality attributes of application framework. Then, we provide explanation of each refactoring method and its positive effects on the application framework quality attributes. Finally, we summarize the refactorings improvements on the application framework quality attributes.

4.1 Selected Refactorings

We select a list of refactorings consisting of 23 refactoring methods which contribute positively to quality attributes of the application framework. We rely on previous studies to determine positive effects of the refactoring methods on the quality attributes. However, some of these refactorings may affect the performance of the application which is not a core design attribute of framework as the framework causes performance degradation due to the additional overhead of dynamic invoking of methods [1]. The refactoring list covers all the categories of the refactorings in different structural levels: package, class, method, field, if-clause and iterative loop. These refactoring methods are among the most popular and most widely used methods.

4.1.1 Add Parameter

Add Parameter is a very common refactoring that is used when a method needs more information from its caller which can be passed by an object [2, 34]. When a method requires information that was not passed in before, then it needs to be changed by adding a parameter [2]. Adding parameter

reduces application-specific details and makes the application more flexible and extensible [1]. Therefore, the refactoring Add Parameter improves the flexibility and extensibility of the application framework.

4.1.2 Decompose Conditional

It is an extraction of methods from complicated conditions statements [2, 34]. Complex conditional logic is one of a common complexity areas in software [2]. This refactoring decomposes the complex conditional logic into a method and replaces its chunks of code with the method call [2]. Decompose conditional is an example of decomposing complex algorithms as it is easy to maintain [28] as well as the decomposed method can be reused [15]. Therefore, the refactoring "Decompose Conditional" improves the reusability and maintainability of the application frameworks.

4.1.3 Encapsulate Field

When there is a public field in a class, it is changed to private and accessors are provided [2, 34]. It keeps the data and its behavior clustered together such that it makes the code easy to maintain, more modular [2] and so more reusable [23]. Therefore, the refactoring "Encapsulate Filed" improves the reusability, modularity and maintainability of the application framework.

4.1.4 Extract Method

It can be implemented by grouping a code fragment together into a method with a name that explains its purpose [2, 34]. Method extraction positively affects maintenance [15] because it simplifies the code by decomposing large methods into simple ones [15, 28]. It also creates new methods which can be reused [15]. Thus, Extract Method improves the reusability and maintainability of the application framework.

4.1.5 Extract Package

A package with too many classes and not easily understandable can be extracted into sub-packages based on dependencies or usages [34]. Class packaging makes the code more flexible because it makes classes' dependencies more explicit [34]. It only concentrates on packaging classes together; therefore, it does not change the internal structures of classes, methods and fields. Thus, the refactoring “Extract Package” improves the flexibility of the application framework.

4.1.6 Extract Subclass

When a certain class contains a subset of features or members only used in some instances, a subclass is created for that subset [2, 34]. Subclass extraction is an implementation of inheritance and abstraction which are essential architecture designs of frameworks [1, 3, 4, 35]. The class abstraction provides more reusability [25]. Subclassing also reduces application-specific details in the source class and makes it more flexible and extensible [1]. Therefore, Extract Subclass improves the reusability, flexibility and extensibility of the application framework.

4.1.7 Extract Super-Class

It is the extraction of a super-class from a set of classes with similar features or members to contain the common features and members [2, 34]. The extracted subclass is defined to be inherited by the source class. It has more general behavior than the behavior of its subclass. Super-class extraction is an implementation of inheritance, generalization and abstraction architecture design of frameworks [1, 3, 4, 35], which provides more reusability [25]. It also includes subclasses creation which provides more flexible and extensible application because it reduces application-specific details in the super-class [1]. Thus, Extract Super-Class improves the reusability, flexibility and extensibility of the application framework.

4.1.8 Hide Delegate

When a client calls a delegate class of an object, methods are created on the server to hide the delegate [2, 34]. This refactoring provides objects encapsulation [2]. The modularity [1] and reusability [36] of the application are enhanced by encapsulation. It also facilitates maintenance because it limits the changes to the server and doesn't require propagation to the client [2]. Thus, the refactoring "Hide Delegate" improves the reusability, modularity and maintainability of the application framework.

4.1.9 Inline Class

When a class is not doing very much, all its features are moved into another class and that class is deleted [2, 34]. The target class is more reusable as it contains more methods which can be reused [15]. Consequently, the refactoring "Inline Class" improves the reusability of the application framework.

4.1.10 Move Field

If a field in a class is used by another class more than its class, a new field is created in the other class and all its users are changed to the new field [2, 34]. When a field is located in a class which uses the field more than other classes, the cohesion of the classes is increased and the coupling among the classes is decreased [23, 26]. Therefore, Move Field improves the reusability and modularity [23].

4.1.11 Parameterize Method

When there are several methods do similar things but with different values contained in the method body, a new method is created using a parameter for the different values [2, 34]. Parameterization provides more flexible [1, 2] and extensible [1] application as it reduces application-specific details [1] and removes duplicate code [2]. Thus, the refactoring "Parameterize Method" improves the flexibility and extensibility of the application framework.

4.1.12 Pull Up Field

When two or more subclasses have same fields, the fields are moved to the super-class [2, 34]. It develops the inheritance and abstraction, essential design features of frameworks [1, 3, 4, 35] and hence provides more reusability [25]. Thus, the refactoring "Pull Up Field" can be used to improve the reusability of the application framework.

4.1.13 Pull Up Method

When a set of subclasses have methods with identical results, the methods are moved to the super-class [2, 34]. This refactoring reduces method duplication among classes and effort for alteration and maintenance by doing generalization and abstraction [2]. It develops generalization, inheritance and abstraction, which are essential architecture designs of frameworks [1, 3, 4, 35] and then provides more reusability [25]. Therefore, Pull Up Method can be used to improve the reusability and maintainability of the application framework.

4.1.14 Push Down Field

When a field in a super-class is used only by some subclasses, the field is moved to those subclasses [2, 34]. It improves the inheritance and abstraction, fundamental design features of frameworks [1, 3, 4, 35] and provides more reusability [25]. Thus, Push Down Field can be used to improve the reusability of the application framework.

4.1.15 Remove Assignments to Parameters

A temporary variable is used instead of assignment to a parameter [2, 34]. It limits parameters to only represent the passed data to the method [2]. It also reduces confusion and provides consistency within the method body code [2] to be more maintainable [28]. Therefore, Remove Assignments to Parameters improves the maintainability of the application framework.

4.1.16 Remove Parameter

A parameter, not used by method body, is removed [2, 34]. This refactoring reduces the number of parameters to be passed in method calling and hence the method becomes easy to reuse [2]. Thus, Remove Parameter improves the reusability of the application framework.

4.1.17 Remove Setting Method

If a field is set at creation time and is never altered, any setting method of that field is removed [2, 34]. This refactoring keeps the necessary setting methods [2], which have high reuse potential, and hence provides more reusability [23]. Thus, Remove Setting Method improves the reusability of the application framework.

4.1.18 Rename Method

If the name of a method does not reveal its purpose, its name should be changed [2, 34]. However, the refactoring Rename Method can be generalized to rename variable, field, class, interface or package. This refactoring makes methods names meaningful, self-explained and unambiguous and hence more maintainable [28] and usable [30]. Therefore, the refactoring "Rename Method" improves the maintainability and usability of the application framework.

4.1.19 Replace Conditional with Polymorphism

If a condition chooses a different behavior based on the type of an object, each clause of the condition is moved to an overriding method in a subclass and the original method is converted to abstract [2, 34]. This refactoring is an implementation of inheritance and abstraction, essential architecture designs of frameworks [1, 3, 4, 35], to make the application more reusable [25], flexible and extensible [2]. It also reduces efforts for maintenance and update and decreases dependencies among components when adding new type to the conditional [2]. Thus, the refactoring "Replace

Conditional with Polymorphism" improves the reusability, maintainability, flexibility and extensibility of the application framework.

4.1.20 Replace Delegation with Inheritance

When many simple delegations are used for the entire interface, then the delegating class can be extracted as a subclass of the delegate [2, 34]. This refactoring implements inheritance and abstraction, essential architecture designs of frameworks [1, 3, 4, 35] and makes the application more reusable [25], flexible and extensible [2]. Therefore, Extract Subclass improves the reusability, flexibility and extensibility of the application framework.

4.1.21 Replace Magic Number with Symbolic Constant

If a literal number with a particular meaning exists, a constant is created with a meaningful name to replace the number [2, 34]. It facilities exploring the logic and provides a great improvement in readability [2] and hence the maintainability [29]. Consequently, Replace Magic Number with Symbolic Constant improves the reusability and maintainability of the application framework.

4.1.22 Reverse Conditional

A condition, which would be easier to understand when it is reversed, should be reversed and reordered [34]. This refactoring allows to re-phrase conditional statements in order to be more readable and understandable [34] and hence more maintainable [28] and usable [30]. Thus, Reverse Conditional improves the maintainability and usability of the application framework.

4.1.23 Split Loop

If a loop does two jobs, the loop is duplicated per job. This refactoring makes the loop blocks clearer [34]. It is an example of decomposing complex algorithms to be maintainable [28]. Therefore, the refactoring "Split Loop" improves the maintainability of the application frameworks. Table 1 shows

the refactorings methods verses the quality attributes of application frameworks. The sign (√) is placed beside any refactoring method under quality attributes positively affected by the refactoring method. The empty cells represent that neither positive nor negative effects investigated.

Table 1 – Refactorings and the Positive Effects on Quality

Refactoring Method	Reusability	Modularity	Maintainability	Usability	Flexibility	Extensibility
1. Add Parameter					√	√
2. Decompose Conditional	√		√			
3. Encapsulate Field	√	√	√			
4. Extract Method	√		√			
5. Extract Package					√	
6. Extract Subclass	√				√	√
7. Extract Super-Class	√				√	√
8. Hide Delegate	√	√	√			
9. Inline Class	√					
10. Move Field	√	√				
11. Parameterize Method					√	√
12. Pull Up Field	√					
13. Pull Up Method	√		√			
14. Push Down Field	√					
15. Remove Assignments to Parameters			√			
16. Remove Parameter	√					
17. Remove Setting Method	√					
18. Rename Method			√	√		
19. Replace Conditional with Polymorphism	√		√		√	√
20. Replace Delegation with Inheritance	√				√	√
21. Replace Magic Number with Symbolic Constant	√		√			
22. Reverse Conditional			√	√		
23. Split Loop			√			

V. Refactoring to Framework

In this section, we present two approaches for refactoring to framework. The objective of these two approaches is to produce domain classes from existing applications to be used as framework in building other applications. The first approach is based on quality attributes of application framework. The second approach is composed of a sequence of ordered refactoring methods and is based on the refactoring level. These approaches are applied using two software applications and are also compared with each other. As an experimental procedure, we perform each refactoring to framework approach on two different applications from different domains and with different project

size. These applications are JAVA Encryption Algorithm (SJEA) [37] and JAVA File Transfer Protocol Client (JFTP) [38]. SJEA is a simple command-line binary encryption algorithm of files (symmetric block cipher) written in JAVA. It uses a password and a byte-vector array to scramble the input file [37, 39] (http://sourceforge.net/projects/sjea/). SJEA application consists of 3 classes. On the other hand JFTP is a simple cross-platform ftp-client coded in JAVA with a command-line interface (CLI) and capable to support Graphical User Interface (GUI) (http://sourceforge.net/projects/javaftp/) [38, 39]. JFTP application consists of 23 classes.

To demonstrate the changes done on each application during the phases of refactoring to framework in terms of size and complexity; system-level results are presented using the software metrics: Line of Code (LOC), Number of Local Methods (NLM), Number of Classes (NOCL) and Number of Packages (NOP). LOC counts total number the physical lines of codes, NLM is the total number of methods with different scopes including private, protected, default and public. NOCL represents the number of classes of the software application and NOP is the number of packages in the software applications. To automate the process of software metrics analysis, we used “Metamata” metrics tool to collect the software metrics for the source code among the phases [40].

5.1 Refactoring to Domain

Domain classes are the classes that implement and provide functionality of a certain domain area while application classes implement the specific application functionality and behavior. To facilitate the refactoring to framework process and prepare the application's classes to be ready for refactoring to framework we apply Refactoring to Domain (RtD) method. The purpose of RtD is to refactor existing classes of applications to domain classes while enhancing their object-oriented features such as inheritance and abstraction.

Refactoring to Domain is done by separating domain and application classes. Therefore, when there is a class that contains both domain and application functionality, a new class is created to contain

the application functionality and the source class becomes a domain class. The domain functionality provides a common core service for the domain while the application functionality provides a specific-application behavior. If the domain functionality of a class is defined as a static method, the static method is replaced with a dynamic one to support object-oriented abstraction which is an essential architecture design of frameworks [1, 3, 4, 35]. In order to do that, the common parameters of the static method are removed and replaced with local variables and their references are replaced too. Constructors with different signatures are created according to the new local variables. Instances of the objects are used in replacing the references of the removed parameters of the static method. If a domain class is final class, the final keyword is removed to allow class extension or inheritance.

5.1.1 Refactoring SJEA to Domain

The SJEA application originally consists of three classes; Checksum, Decryption and Encryption. All of the three classes contain domain and application functionalities. The domain functions are the methods responsible for creating checksum, decryption and encryption. The application functions are the main methods in each class providing the command-line interface of each class.

To refactor SJEA to domain, we first create a new class for the application functions (main methods) in order to separate the application class from the domain classes. Consequently, the domain classes SJEA are Checksum, Decryption and Encryption while the application class is SJEAApplication. Since the domain functions of the SJEA classes are originally implemented as static methods, they are converted to dynamic methods. To perform this, the common parameter of the static methods (file) is replaced with a local variable and two constructors are created. As a result of applying refactoring to domain, the SJEA application is ready to be refactored to framework.

5.1.2 Refactoring JFTP to Domain

The JFTP design distinguishes and separates the domain classes from the application classes. However, it has a domain class named FTPCmdServer that includes all FTP command services functions. This class is final. The final class in java cannot be inherited or extended. Therefore, the final keyword is removed as a step of refactoring to domain, as a result, the JFTP application becomes ready to be refactored to framework.

5.2 Quality Attribute Based Refactoring To Framework (QARtF)

In this section, we propose a quality attribute based approach refactoring to framework (QARtF). It can be followed to improve the domain classes of an existing application to be used as framework. QARtF focuses on improving the framework quality attributes of an application.

5.2.1 Quality Attribute Based Refactoring To Framework Process

QARtF process consists of three ordered phases: reusability and modularity, maintainability and usability and flexibility and extensibility. The process merges the reusability and modularity phase because they include a large set of refactoring methods and also the reusability is improved when the modularity is improved [23]. Then, it merges the maintainability and usability in one phase because they share the same refactoring methods. It merges the flexibility and extensibility in the last phase because they cover same set of refactorings. We structure the phases based on the number of refactoring methods per phase in descending order. The reason of this structuring order is to apply maximum number of refactoring methods and to cover overlapping refactorings in early phases of the approach. As a suggested recommendation, meaningful code statements and clear comments should be used in each step of the phases of QARtF. However, the refactoring methods of each phase of QARtF are not ordered.

- **Reusability and Modularity Refactoring**

Reusability and modularity refactoring is the first phase of QARtF approach. It encompasses refactoring methods that positively affect the reusability and modularity of an application. There are 16 refactorings in the usability and modularity phase as shown in Table 1.

- **Maintainability and Usability Refactoring**

Maintainability and usability refactoring is the second phase of QARtF approach. It covers refactorings that improve the maintainability and usability of an application. There are 11 refactorings methods in the maintainability and usability phase as shown in Table 1.

- **Flexibility and Extensibility Refactoring**

Flexibility and extensibility refactoring is the last phase of QARtF approach. It includes a set of refactorings that positively affect the flexibility and extensibility of an application. There are 7 refactoring methods in this phase as shown in Table 1.

- **Meaningful Coding and Clear Commenting**

The maintainability of an application is improved when its source code has meaningful components identification and names and understandable statement, commands and understandable comments [28]. This step is applicable in any phase of QARtF. Rename Method, Field, Variable, Class, Interface and Package refactorings are useful in this step.

5.2.2 Refactoring SJEA to Framework Using QARtF

The SJEA application is refactored to framework using the QARtF approach in three phases. First, the reusability and modularity refactorings are applied on each bad smells of the SJEA application (bad smells are symptom or warning signs in the source code of the program that possibly indicates a potential problem, these bad smells are indicators that the code should be refactored). Second, all of applicable maintainability and usability refactorings are made on the SJEA classes. Finally, the

flexibility of the SJEA is improved by applying the last phase of QARtF. In each phase QARtF, meaningful coding and clear commenting step is used.

In refactoring SJEA to framework, most of the refactoring methods of the phases of QARtF are applied. Table 2 shows the refactoring methods that are applied on SJEA system.

Table 2 – SJEA Classes Refactoring Using QARtF

Class	Type	Refactorings Applied (Phase Number)
Checksum	Domain	Encapsulate Field (1), Remove Setting Method (1), Replace Magic Number with Symbolic Constant (1), Extract Method (1), Extract Package (3)
ChecksumMD5	Domain	Extract Subclass (1), Extract Package (3)
Cryptography	Domain	Extract Super-Class (1), Pull Up Method (1), Replace Delegation with Inheritance (1), Extract Package (3)
Decryption	Domain	Encapsulate Field (1), Remove Setting Method (1), Hide Delegate (1), Extract Method (1), Extract Package (3)
Encryption	Domain	Encapsulate Field (1), Remove Setting Method (1), Hide Delegate (1), Extract Method (1), Extract Package (3)
SJEAApplication	Application	N/A

The results, in terms of system-level LOC, NLM, NOCL and NOP, of each phase of the QARtF approach are presented in Table 3.

Table 3 – Results of SJEA Refactoring Using QARtF

Phase	LOC	NLM	NOCL	NOP
SJEA Application	334	7	4	1
Reusability and Modularity of SJEA	328	11	6	1
Maintainability and Usability of SJEA	328	11	6	1
Flexibility and Extensibility of SJEA	331	11	6	3
SJEA Framework	331	11	6	3

5.2.3 Refactoring JFTP to Framework Using QARtF

The JFTP application is refactored to framework via the phases of the QARtF approach. First, the reusability and modularity refactorings are applied on each bad smells of the JFTP application. Second, all of applicable maintainability and usability refactorings are applied on the JFTP classes. Finally, the flexibility and extensibility of the JFTP are improved by applying the flexibility and extensibility refactorings. In each phase of QARtF, meaningful coding and clear commenting step is

used. In refactoring JFTP to framework, all of the refactoring methods of the QARtF phases are applied. Table 4 shows the refactoring methods applies on JFTP system.

Table 4 – JFTP Classes Refactoring Using QARtF

Class	Type	Refactorings Applied (Phase Number)
Authenticateable	FTP Domain	Inline Class (1), Extract Package (3)
CommandLineParser	FTP Domain	Decompose Conditional (1), Hide Delegate (1), Remove Assignments to Parameters (2), Rename Method (2), Add Parameter (3), Extract Package (3)
Connectable	FTP Domain	Inline Class (1), Extract Package (3)
Createable	FTP Domain	Inline Class (1), Extract Package (3)
DataTypeChangeable	FTP Domain	Extract Subclass (1), Inline Class (1), Extract Package (3)
DirectoryChangeable	FTP Domain	Extract Subclass (1), Inline Class (1), Extract Package (3)
FTPASCIIData	FTP Domain	Decompose Conditional (1), Replace Conditional with Polymorphism (1), Extract Package (3)
FTPAuthentication	FTP Domain	Decompose Conditional (1), Extract Subclass (1), Push Down Field (1), Extract Package (3)
FTPBinaryData	FTP Domain	Decompose Conditional (1), Replace Conditional with Polymorphism (1), Extract Package (3)
FTPCmdServer	FTP Domain	Encapsulate Field (1), Remove Setting Method (1), Extract Package (3)
FTPConnection	FTP Domain	Decompose Conditional (1), Extract Method (1), Extract Subclass (1), Extract Super-Class (1), Hide Delegate (1), Pull Up Method (1), Push Down Field (1), Remove Parameter (1), Remove Assignments to Parameters (2), Extract Package (3), Parameterize Method (3)
FTPDataType	FTP Domain	Extract Method (1), Extract Subclass (1), Replace Magic Number with Symbolic Constant (1), Extract Package (3)
FTPDirectory	FTP Domain	Decompose Conditional (1), Extract Subclass (1), Hide Delegate (1), Push Down Field (1), Reverse Conditional (2), Split Loop (2), Extract Package (3)
FTPTransfer	FTP Domain	Decompose Conditional (1), Extract Subclass (1), Hide Delegate (1), Push Down Field (1), Replace Delegation with Inheritance (1), Extract Package (3)
JFTP	Application	Decompose Conditional (1), Extract Method (1), Extract Subclass (1), Move Field (1), Remove Assignments to Parameters (2)
JFTPSuper	FTP Domain	Extract Package (3)
Listable	FTP Domain	Inline Class (1), Extract Package (3)
NetIO	IO Domain	Encapsulate Field (1), Inline Class (1), Pull Up Field (1), Pull Up Method (1), Extract Package (3)
NetReader	IO Domain	Extract Package (3)
NetWriter	IO Domain	Move Field (1), Extract Package (3)
Parser	FTP Domain	Extract Package (3)
Progressbar	Application	Extract Package (3)
Removeable	FTP Domain	Inline Class (1), Extract Package (3)
ServerResponseParser	FTP Domain	Extract Package (3)
StdErr	IO Domain	Inline Class (1), Extract Package (3)
StdIn	IO Domain	Inline Class (1), Extract Package (3)
StdOut	IO Domain	Inline Class (1), Move Field (1), Extract Package (3)
Transferable	FTP Domain	Inline Class (1), Extract Package (3)

The results, in terms of system-level LOC, NLM, NOCL and NOP, of each phase of the QARtF approach are presented in Table 5

Table 5 – Results of JFTP Refactoring Using QARtF

Phase	LOC	NLM	NOCL	NOP
JFTP Application	1490	180	23	1
Reusability and Modularity of JFTP	1711	256	28	1
Maintainability and Usability of JFTP	1817	281	28	1
Flexibility and Extensibility of JFTP	1865	283	28	6
JFTP Framework	1865	283	28	6

5.3 Level-Based Refactoring to Framework (LRtF)

In this section, we propose a level-based approach of refactoring to framework (LRtF).

5.3.1 Level-Based Refactoring to Framework Process

LRtF process consists of a sequence of ordered refactoring methods. The refactoring methods of the level-based refactoring to framework include all refactorings of the quality attribute based refactoring to framework (QARtF). It covers all refactorings which improve and enhance the application framework quality attributes. In the level-based refactoring to framework, we classify and organize the refactorings in a sequential approach. They are classified into five different levels that are applied in the following sequence: class level, package level, field level, method level and block level. The block level is divided into if-clause level and loop level. The level classification identifies the type of changes that a refactoring does on the source code. It can be used to avoid overlap and conflict between the refactoring methods. Table 6 summarizes the steps of the level-based refactoring to framework (LRtF); it lists the order of refactoring methods that should be applied to refactor to framework.

Table 6 – Level-Based Refactoring to Framework (LRtF)

	Refactoring	Level
1	Extract Subclass	Class
2	Extract Super-Class	Class
3	Replace Conditional with Polymorphism	Class
4	Hide Delegate	Class
5	Replace Delegation with Inheritance	Class
6	Inline Class	Class

7	Extract Package	Package
8	Replace Magic Number with Symbolic Constant	Field
9	Encapsulate Field	Field
10	Push Down Field	Field
11	Move Field	Field
12	Pull Up Field	Field
13	Extract Method	Method
14	Decompose Conditional	Method
15	Remove Setting Method	Method
16	Pull Up Method	Method
17	Remove Parameter	Method
18	Add Parameter	Method
19	Parameterize Method	Method
20	Remove Assignments to Parameters	Method
21	Rename Method	Method
22	Reverse Conditional	If Clause
23	Split Loop	Loop

- **Class-Level Refactorings**

The class-level refactorings are applied first because classes determine the size of the application in terms of number of classes in early stage of the refactoring to framework. The class-level refactorings are performed according to extract refactorings, abstract refactorings, delegate refactorings and remove refactorings. First, All possible subclasses are created using Extract Subclass. Extract Subclass because some of the extracted subclasses can be refactored using next class-level refactorings. Then, the common features of classes can be combined in a super-class using Extract Super-Class refactoring. Extract Super-Class refactoring can be applied on the refactored subclasses resulted by Extract Subclass. Next, the abstract refactoring is applied using Replace Conditional with Polymorphism method. The type conditional expression is changed to polymorphism using the refactoring method Replace Conditional with Polymorphism. This improves inheritance and abstraction levels of the application. Hide Delegate and Replace Delegation with Inheritance are delegate refactorings and applied after abstract refactoring. This is to finalize the inheritance and abstraction level of the classes. Hide Delegate is used to hide all delegates of classes. Then, Replace Delegation with Inheritance refactoring method is used on existing delegates and hidden delegates to be replaced by inheritance. This increases the level of abstraction and inheritance. Finally at class-level refactoring, all unnecessary classes and interfaces are removed through Inline

Class refactoring method. The class-level refactorings fix the size of the application in terms of the number of classes and finalize the level of abstraction and inheritance.

- **Package-Level Refactorings**

Since the application size is fixed and the number of classes and interfaces are determined in the class-level refactorings phase, it is easy to group classes into different scopes or packages based on their functionalities. Different packages can be extracted using the refactoring Extract Package.

- **Field-Level Refactorings**

The field-level refactorings are done before the method-level refactoring because some field-level refactorings cause creating new methods and these new methods may need to be refactored using the method-level refactorings. The field-level refactoring starts with Replace Magic Number with Symbolic Constant because the symbolic constant can be replaced by a field in the class which may need some other field-level refactorings. Then, all fields of the class are encapsulated in getter and setter methods via the refactoring method Encapsulate Field. After field encapsulation, the specialized fields of a super-class are pushed down with their encapsulations to appropriate subclasses using Push Down Field. Move Field refactoring is useful to transfer certain fields with their encapsulations from a class to another. The common original or refactored fields are pulled up with their encapsulations to a super-class by the refactoring method Pull Up Field.

- **Method-Level Refactorings**

The method-level refactorings begin with Extract Method as a kind of decomposing complex algorithms. Complex conditional is extracted to a new method using Decompose Conditional refactoring. Unnecessary encapsulation methods are removed through the refactoring method Remove Setting Method. Then, the exiting, extracted or decomposed methods are moved from a subclass to a super-class using Pull Up Method refactoring. The external structures of classes are

finalized by the refactoring Pull Up Method. After that, all needless parameters of methods are removed by the refactoring Remove Parameter. Add Parameter refactoring is used to add parameters to methods to increase the flexibility of the methods. Sometimes, a method is more useful if it is overloaded with different signature to have parameters using the refactoring Parameterize Method. However, method parameters should be read only by applying Remove Assignments to Parameters refactoring. Rename Method refactoring is used to make the method reveal its purpose. The method-level refactorings finalize and fix total number of methods of an application.

- **If-Clause-Level Refactorings**

The block-level refactoring comes after the method-level refactoring because the method-level refactoring finalizes the number of methods of classes. The if-clause-level refactorings should come before the loop-level refactoring. This is because the loop-level results code duplication because it splits a single loop into multiple loops. If the split loop contains a condition that needs some if-clause refactorings, then starting with the if-clause-level refactorings reduces the amount of refactoring work at loop-level refactoring. Therefore, it is better to do if-clause-level refactorings before loop-level refactorings. When a condition expression is needed to be reversed to be more readable and understandable, the if-clause-level refactoring Reverse Conditional is applied.

- **Loop-Level Refactorings**

The loop-level refactoring is performed after the if-clause-level refactoring. Split Loop refactoring method is used to decompose a complex loop into several duplicated simple loops.

- **Name and Comment Refactorings**

The naming refactoring methods can be used in any phase of the level-based refactoring to framework without any negative effects on other refactorings. The naming refactorings includes Rename Method, Rename Variable, Rename Field, Rename Class, Rename Interface and Rename

Package. The main advantage of the naming refactoring is to make method, variable, field, class, interface, or package more readable and more understandable. In addition, meaningful comments on source code statements, explaining the input and output specifications, help and improve the readability of the source code statements. They can be done using Add Comments. However, this phase can be done during all other phases of LRtF.

5.3.2 Refactoring SJEA to Framework Using LRtF

The SJEA application is refactored to framework using the LRtF approach. First, the class-level refactorings are applied on each bad smells of the SJEA application. Then, the package-level refactoring is made on the SJEA classes. The field-level refactorings, after that, are applied on the SJEA application. Some method-level refactorings are also done on the SJEA classes. In each phase LRtF, meaningful coding and clear commenting are used.

In refactoring SJEA to a framework, most of the refactoring methods of the phases of LRtF are applied. The class-level refactorings include Extract Subclass, Extract Super-Class, Hide Delegate and Replace Delegation with Inheritance. The package-level refactoring is Extract Package. The field-level refactorings applied are Encapsulate Field, Pull Up Field and Replace Magic Number with Symbolic Constant. The method-level refactorings applied include Extract Method, Pull Up Method and Remove Setting Method. The SJEA application does not have bad smells for the if-clause-level and loop-level refactorings Table 7 shows the refactoring methods of LRtF applied on SJEA classes.

Table 7 – SJEA Classes Refactoring Using LRtF

Class	Type	Refactorings Applied (Phase Number)
Checksum	Domain	Encapsulate Field (3), Remove Setting Method (4), Replace Magic Number with Symbolic Constant (3), Extract Method (4), Extract Package (2)
ChecksumMD5	Domain	Extract Subclass (1), Extract Package (2)
Cryptography	Domain	Extract Super-Class (1), Pull Up Method (4), Replace Delegation with Inheritance (1), Extract Package (2)
Decryption	Domain	Encapsulate Field (3), Remove Setting Method (4), Hide Delegate (1), Extract Method (4), Extract Package (2)
Encryption	Domain	Encapsulate Field (3), Remove Setting Method (4), Hide Delegate (1), Extract Method (4), Extract Package (2)
SJEAApplication	Application	N/A

The system-level results, in terms of LOC, NLM, NOCL and NOP, of each phase of the LRtF approach are presented in Table 8. It shows that there is no change accomplished in if-clause-level and loop-level phases because there are no suitable bad smells in the SJEA. It presents increment in NLM, NOCL and NOP. NLM is increased due to methods extraction during LRtF phases. NOCL is increased because of classes' extraction as new subclasses and super-classes are created. NOP is increased because packages are extracted to group relevant classes together.

Table 8 – Results of SJEA Refactoring Using LRtF

Phase	LOC	NLM	NOCL	NOP
SJEA Application	334	7	4	1
Class-Level of SJEA	314	5	6	1
Package-Level of SJEA	317	5	6	3
Field-Level of SJEA	331	11	6	3
Method-Level of SJEA	331	11	6	3
If-Clause-Level of SJEA	331	11	6	3
Loop-Level of SJEA	331	11	6	3
SJEA Framework	331	11	6	3

5.3.3 Refactoring JFTP to Framework Using LRtF

The JFTP application is refactored to framework via the phases of the LRtF approach. First, the class-level refactorings are applied on each bad smells of the JFTP application. Then, the package-level refactoring is made on the JFTP classes. The field-level refactorings, after that, are applied on the JFTP application. The method-level refactorings are also done on the JFTP classes. Finally, the block-level refactorings (if-clause-level and loop-level) are applied. Meaningful coding and clear commenting are used while refactoring the JFTP application in each phase LRtF.

In refactoring JFTP to a framework using LRtF, all of the refactoring methods of the LRtF phases are applied. The class-level refactorings include Inline Class, Extract Subclass, Extract Super-Class, Replace Conditional with Polymorphism, Hide Delegate and Replace Delegation with Inheritance. The package-level refactoring is Extract Package. The field-level refactorings applied are Replace Magic Number with Symbolic Constant, Encapsulate Field, Push Down Field, Move Field and Pull Up Field. The method-level refactorings applied includes Extract Method, Decompose Conditional, Remove Setting Method, Pull Up Method, Remove Parameter, Add Parameter, Parameterize Method, Remove Assignments to Parameters, and Rename Method. The loop-level refactoring applied is Split Loop and the if-clause-level refactoring applied is Reverse Conditional. Table 9 shows the refactoring methods of LRtF applied on JFTP classes.

Table 9 – JFTP Classes Refactoring Using LRtF

Class	Type	Refactorings Applied (Phase Number)
Authenticateable	FTP Domain	Inline Class (1), Extract Package (2)
CommandLineParser	FTP Domain	Decompose Conditional (4), Hide Delegate (1), Remove Assignments to Parameters (4), Rename Method (4), Add Parameter (4), Extract Package (2)
Connectable	FTP Domain	Inline Class (1), Extract Package (2)
Createable	FTP Domain	Inline Class (1), Extract Package (2)
DataTypeChangeable	FTP Domain	Extract Subclass (1), Inline Class (1), Extract Package (2)
DirectoryChangeable	FTP Domain	Extract Subclass (1), Inline Class (1), Extract Package (2)
FTPASCIIData	FTP Domain	Decompose Conditional (4), Replace Conditional with Polymorphism (1), Extract Package (2)
FTPAuthentication	FTP Domain	Decompose Conditional (4), Extract Subclass (1), Push Down Field (3), Extract Package (2)
FTPBinaryData	FTP Domain	Decompose Conditional (4), Replace Conditional with Polymorphism (1), Extract Package (2)
FTPCmdServer	FTP Domain	Encapsulate Field (3), Remove Setting Method (4), Extract Package (2)
FTPConnection	FTP Domain	Decompose Conditional (4), Extract Method (4), Extract Subclass (1), Extract Super-Class (1), Hide Delegate (1), Pull Up Method (4), Push Down Field (3), Remove Parameter (4), Remove Assignments to Parameters (4), Extract Package (2), Parameterize Method (4)
FTPDataType	FTP Domain	Extract Method (4), Extract Subclass (1), Replace Magic Number with Symbolic Constant (3), Extract Package (2)
FTPDirectory	FTP Domain	Decompose Conditional (4), Extract Subclass (1), Hide Delegate (1), Push Down Field (3), Reverse Conditional (5), Split Loop (6), Extract Package (2)
FTPTransfer	FTP Domain	Decompose Conditional (4), Extract Subclass (1), Hide Delegate (1), Push Down Field (3), Replace Delegation with Inheritance (1), Extract Package (2)

JFTP	Application	Decompose Conditional (4), Extract Method (4), Extract Subclass (1), Move Field (3), Remove Assignments to Parameters (4)
JFTPSuper	FTP Domain	Extract Package (2)
Listable	FTP Domain	Inline Class (1), Extract Package (2)
NetIO	IO Domain	Encapsulate Field (3), Inline Class (1), Pull Up Field (3), Pull Up Method (4), Extract Package (2)
NetReader	IO Domain	Extract Package (2)
NetWriter	IO Domain	Move Field (3), Extract Package (2)
Parser	FTP Domain	Extract Package (2)
Progressbar	Application	Extract Package (2)
Removeable	FTP Domain	Inline Class (1), Extract Package (2)
ServerResponseParser	FTP Domain	Extract Package (2)
StdErr	IO Domain	Inline Class (1), Extract Package (2)
StdIn	IO Domain	Inline Class (1), Extract Package (2)
StdOut	IO Domain	Inline Class (1), Move Field (3), Extract Package (2)
Transferable	FTP Domain	Inline Class (1), Extract Package (2)

The system-level results are presented in terms of LOC, NLM, NOCL and NOP for each phase of the LRtF approach in Table 10. It illustrates the changes of the JFTP application accomplished in each phase of LRtF. It shows increment occurred on NLM, NOCL and NOP. NLM is increased due to methods extraction and decomposition during the refactoring to framework phases. NOCL is increased because of classes' extraction and abstraction as new subclasses and super-classes are created. NOP is increased because packages are extracted to group related classes together.

Table 10 – Results of JFTP Refactoring Using LRtF

Phase	LOC	NLM	NOCL	NOP
JFTP Application	1490	180	23	1
Class-Level of JFTP	1577	191	28	1
Package-Level of JFTP	1611	191	28	6
Field-Level of JFTP	1754	223	28	6
Method-Level of JFTP	1861	283	28	6
If-Clause-Level of JFTP	1861	283	28	6
Loop-Level of JFTP	1865	283	28	6
JFTP Framework	1865	283	28	6

We notice from Table 11 and Table 12 that the two approaches of refactoring generate the same set of domain and application classes.

Table 11 – SJEA Classes after Refactoring to Framework

Class	Type	Description
Checksum	Domain	A Class implements checksum creation
ChecksumMD5	Domain	A Class implements checksum creation using MD5 algorithm
Cryptography	Domain	A Class implements cryptography of MD5 algorithm
Decryption	Domain	A Class implements decryption using MD5 algorithm
Encryption	Domain	A Class implements encryption using MD5 algorithm
SJEAApplication	Application	A Class implements the application user interface operation

Table 12 – JFTP Classes after Refactoring to Framework

Class	Type	Description
Connectable	FTP Domain	An Interface includes methods for FTP connection
Authenticateable	FTP Domain	An Interface includes methods for FTP authentication
Listable	FTP Domain	An Interface includes methods for FTP directory list
DataTypeChangeable	FTP Domain	An Interface includes methods for FTP data mode change
DirectoryChangeable	FTP Domain	An Interface includes methods for FTP directory change
Createable	FTP Domain	An Interface includes methods for FTP directory creation
Removeable	FTP Domain	An Interface includes methods for FTP directory remove
Transferable	FTP Domain	An Interface includes methods wrapping all FTP data transfer
JFTPSuper	FTP Domain	A Super-Class of all FTP activities classes
FTPCmdServer	FTP Domain	A Super-Class of FTP activities and commands classes
FTPConnection	FTP Domain	A Class implements FTP connection activities
FTPAuthentication	FTP Domain	A Class implements FTP authentication activities
FTPDataType	FTP Domain	A Class implements FTP data mode change activities
FTPASCIIData	FTP Domain	A Class implements FTP data type change to ASCII
FTPBinaryData	FTP Domain	A Class implements FTP data type change to Binary or Image
FTPDirectory	FTP Domain	A Class implements FTP directory manipulation activities
FTPTransfer	FTP Domain	A Class implements FTP data transfer activities
Parser	FTP Domain	A Super-Class of all FTP messages parsing classes
CommandLineParser	FTP Domain	A Class implements FTP command parsing for requests
ServerResponseParser	FTP Domain	A Class implements FTP command parsing for responses
NetIO	IO Domain	A Class implements network IO operations
NetReader	IO Domain	A Class implements network reading operation
NetWriter	IO Domain	A Class implements network writing operation
StdErr	IO Domain	A Class implements standard error reporting operation
StdIn	IO Domain	A Class implements standard input reporting operation
StdOut	IO Domain	A Class implements standard output reporting operation
JFTP	Application	A Class implements the application user interface operation
Progressbar	Application	A Class implements progress bar for interface operation

VI. Conclusion

Refactoring to framework is software refactoring or restructuring to produce reusable domain classes for a specific problem domain that can be used as a common application framework. In this paper we proposed two refactoring to framework approaches consisting of a set of refactoring methods: the quality attribute based refactoring to framework (QARtF) and the level based refactoring to framework (LRtF). The two refactoring to framework approaches provide a standard approach for application frameworks development using refactoring.

We validated the two approaches empirically using two open source systems. We conclude that although QARtF and LRtF share common features, each one also has its own characteristics. Both approaches yield to the same set of domain classes and application classes. They both have the same quality attributes improvement for the application framework at system level. However, QARtF is an ordered phase level approach while LRtF is a sequential ordered level approach. QARtF requires the designer to be experienced with quality attributes. However, LRtF does not rely on the designer's experience in quality attributes as it is a sequence of refactoring methods.

In future work we plan to apply the refactoring to framework approaches in many applications from diverse domain areas in different scales, enhance the refactoring to framework approaches to include more refactoring methods that positively affect quality attributes of framework and develop a refactoring process approach to convert a set of existing applications to a product line.

Acknowledgement

The authors acknowledge the support of King Fahd University of Petroleum and Minerals in the development of this work.

References

[1] M. Fayad and D. Schmidt, "Object-Oriented Application Frameworks," Communications of the ACM, vol. 40, October, 1997 1997.

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, Refactoring: Improving the Design of Existing Code, First ed.: Addison-Wesley Longman Publishing Co., Inc., 1999.

[3] M. Fayad, D. Schmidt, and R. Johnson, Building Application Frameworks: Object-Oriented Foundations of Framework Design, First ed.: John Wiley and Sons, Inc., 1999.

[4] M. Fayad, D. Schmidt, and R. Johnson, Implementing Application Frameworks: Object-Oriented Frameworks at Work, First ed.: John Wiley and Sons, Inc., 1999.

[5] R. Tiarks, "Quality-Driven Refactoring," presented at the Informatic Seminar Transformation (IST'05), University of Bremen, Bremen, Germany, 2005.

[6] L. Tahvildari, "Quality-Driven Object-Oriented Re-Engineering Framework," presented at the The 20th IEEE International Conference on Software Maintenance (ICSM'04), Chicago, IL, USA, 2004.

[7] Z. Xing and S. Eleni, "Refactoring Practice: How it is and How it Should be Supported - An Eclipse Case Study," presented at the The 22nd IEEE International Conference on Software Maintenance (ICSM'06), Philadelphia, Pennsylvania, USA, 2006.

[8] L. Tahvildari, K. Kontogiannis, and J. Mylopoulos, "Quality-Driven Software Re-Engineering," Journal of Systems and Software, vol. 66, pp. 225-239, June, 2003 2003.

[9] B. Geppert, A. Mockus, and F. Rossler, "Refactoring for Changeability: A Way to Go?," presented at the The 11th IEEE International Software Metrics Symposium (METRICS'05), Como, Italy, 2005.

[10] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi, "A Case Study in Refactoring a Legacy Component for Reuse in a Product Line," presented at the The 21st IEEE International Conference on Software Maintenance (ICSM'05), Budapest, Hungary, 2005.

[11] M. Goldstein, Y. A. Feldman, and S. Tyszberowicz, "Refactoring with Contracts," presented at the IEEE Agile Conference (AGILE'06), Minneapolis, Minnesota, USA, 2006.

[12] N. Ubayashi, J. Piao, S. Shinotsuka, and T. Tamai, "Contract-Based Verification for Aspect-Oriented Refactoring," presented at the The 2008 International Conference on Software Testing, Verification, and Validation (ICST'08), Lillehammer, Norway, 2008.

[13] S. Bryton and F. Abreu, "Modularity-Oriented Refactoring," presented at the The 12th European Conference on Software Maintenance and Reengineering (CSMR'08), Athens, Greece, 2008.

[14] S. Shrivastava and V. Shrivastava, "Impact of Metrics Based Refactoring on the Software Quality: a Case Study," presented at the The 2008 IEEE Region 10 Conference (TENCON'08), Hyderabad, India, 2008.

[15] N. Tsantalis and A. Chatzigeorgiou, "Identification of Extract Method Refactoring Opportunities," presented at the The 13th European Conference on Software Maintenance and Reengineering (CSMR'09), Kaiserslautern, Germany, 2009.

[16] I. Ivkovic and K. Kontogiannis, "A Framework for Software Architecture Refactoring using Model Transformations and Semantic Annotations," presented at the The Conference on Software Maintenance and Reengineering (CSMR'06), Bari, Italy, 2006.

[17] L. Grunske and R. Neumann, "Process Components for Quality Evaluation and Quality Improvement," presented at the The 2nd Workshop on Method Engineering for Object-Oriented and Component-Based Development (OOPSLA'04), The International Conference on Object Oriented Programming, Systems, Languages and Applications, 2004.

[18] E. Murphy-Hill and A. Black, "Refactoring Tools: Fitness for Purpose," IEEE Software, vol. 25, pp. 38-44, September-October, 2008 2008.

[19] N. C. Mendonca, P. H. Maia, L. A. Fonseca, and R. M. Andrade, "RefaX: A Refactoring Framework Based on XML," presented at the The 20th IEEE International Conference on Software Maintenance (ICSM'04), Chicago, IL, USA, 2004.

[20] K. Maruyama and S. Yamamoto, "Design and Implementation of an Extensible and Modifiable Refactoring Tool," presented at the The 13th International Workshop on Program Comprehension (IWPC'05), St. Louis, MO, USA, 2005.

[21] R. Marticorena, C. Lopez, Y. Crespo, and F. J. Perez, "Reuse Based Refactoring Tools," presented at the The 1st Workshop on Refactoring Tools (WRT'07), The 21st European Conference on Object-Oriented Programming (ECOOP'07), Berlin, Germany, 2007.

[22] E. Murphy-Hill and A. Black, "Breaking the Barriers to Successful Refactoring: Observations and Tools for Extract Method," presented at the ACM/IEEE 30th International Conference on Software Engineering (ICSE'08), Leipzig, Germany, 2008.

[23] S. Lai and C. Yang, "A Software Metric Combination Model for Software Reuse," presented at the Fifth Asia-Pacific Software Engineering Conference (APSEC'98), Taipei, Taiwan, 1998.

[24] J. Perry, "Perspective on Software Reuse," Software Engineering Institute (SEI), Technical Report CMU/SEI-88-SR-022September, 1988 1988.

[25] J. Estublier and G. Vega, "Reuse and Variability in Large Software Applications," Special Interest Group on Software Engineering (SIGSOFT) Software Engeering Notes, vol. 30, pp. 316-325, September, 2005 2005.

[26] F. Dandashi, "A Method for Assessing the Reusability of Object-Oriented Code Using a Validated Set of Automated Measurements," presented at the The 2002 ACM Symposium on Applied Computing (SAC'02), Madrid, Spain, 2002.

[27] T. Lopes, I. Neag, and J. Ralph, "The Role of Extensibility in Software Standards for Automatic Test Systems," presented at the IEEE Autotestcon (AUTOTESTCON'05), 2005.

[28] F. Zhuo, B. Lowther, P. Oman, and J. Hagemeister, "Constructing and Testing Software Maintainability Assessment Models," presented at the First International Software Metrics Symposium (METRICS'93), Baltimore, Maryland, USA, 1993.

[29] R. Land, "Measurements of Software Maintainability," presented at the Graduate Student Conference, Uppsala University, Uppsala, Sweden, 2002.

[30] N. Bevan, "Measuring Usability as Quality of Use," Journal of Software Quality, vol. 4, pp. 115-130, March, 1995 1995.

[31] N. Bevan and M. Macleod, "Usability Measurement in Context," Behaviour and Information Technology (BIT), vol. 13, pp. 132-145, 1994.

[32] M. Bertoa and A. Vallecillo, "Usability Metrics for Software Components," presented at the Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE'04), The 8th European Conference on Object-Oriented Programming (ECOOP'04), Oslo, Norway, 2004.

[33] A. Eden and T. Mens, "Measuring Software Flexibility," IEE Proceedings Software, vol. 153, pp. 113-126, June, 2006 2006.

[34] M. Fowler. (2008, November 18, 2008). Refactoring Catalog: Refactorings in Alphabetical Order. Available: http://www.refactoring.com/catalog/index.html

[35] R. E. Johnson, "How Frameworks Compare To Other Object-Oriented Reuse Techniques: Frameworks = (Components + Patterns)," Communications of the ACM, vol. 40, October, 1997 1997.

[36] K. Rubin, "Reuse in Software Engineering: an Object-Oriented Perspective," presented at the IEEE COMPCON, 1990.

[37] SJEA. (2009, March 21, 2009). Simple JAVA Encryption Algorithm (SJEA). Available: http://sourceforge.net/projects/sjea/

[38] JFTP. (2001, March 21, 2009). JAVA File Transfer Protocol Client (JFTP). Available: http://sourceforge.net/projects/javaftp/

[39] SourceForge. (2009, March 21, 2009). Source Forge: Find and Build Open Source Software. Available: http://sourceforge.net/index.php

[40] MetaMata. (2000). MetaMata Metrics Tool v2.0. Available: http://www.metamata.com

Mohammad Alshayeb is an associate professor in Software Engineering at the Information and Computer Science Department, King Fahd University of Petroleum and Minerals (KFUPM), Dhahran, Saudi Arabia. He received his MS and PhD in Computer Science and certificate of Software Engineering from the University of Alabama in Huntsville. He worked as a senior researcher and Software Engineer and managed software projects in the United States and Middle East. He is a certified project manager (PMP). His research interests include Software quality, software measurement and metrics, and empirical studies in software engineering.