RESEARCH ARTICLE OPEN ACCESS

DESQA a Software Quality Assurance Framework

Dr.KhaledKh. S. Kh. Allanqawi
General Subjects Department, High Institute of Energy, Public Authority of Applied education and Training, Kuwait

ABSTRACT
In current software development lifecycles of heterogeneous environments, the pitfalls businesses have to face are that software defect tracking, measurements and quality assurance do not start early enough in the development process. In fact the cost of fixing a defect in a production environment is much higher than in the initial phases of the Software Development Life Cycle (SDLC) which is particularly true for Service Oriented Architecture (SOA). Thus the aim of this study is to develop a new framework for defect tracking and detection and quality estimation for early stages particularly for the design stage of the SDLC. Part of the objectives of this work is to conceptualize, borrow and customize from known frameworks, such as object-oriented programming to build a solid framework using automated rule based intelligent mechanisms to detect and classify defects in software design of SOA. The implementation part demonstrated how the framework can predict the quality level of the designed software. The results showed a good level of quality estimation can be achieved based on the number of design attributes, the number of quality attributes and the number of SOA Design Defects. Assessment shows that metrics provide guidelines to indicate the progress that a software system has made and the quality of design. Using these guidelines, we can develop more usable and maintainable software systems to fulfill the demand of efficient systems for software applications. Another valuable result coming from this study is that developers are trying to keep backwards compatibility when they introduce new functionality. Sometimes, in the same newly-introduced elements developers perform necessary breaking changes in future versions. In that way they give time to their clients to adapt their systems. This is a very valuable practice for the developers because they have more time to assess the quality of their software before releasing it. Other improvements in this research include investigation of other design attributes and SOA Design Defects which can be computed in extending the tests we performed.

I. INTRODUCTION

Historically, the software quality management process was focused on finding the defects in software and correcting them. This took place in two steps, developing software to completion and checking for defects in the end product. The shortcoming of this approach was that the same defects would still be realised in another software process [1]. It is important to consider the uniqueness of each piece of software. They are designed as artifacts and meant to serve the user needs adequately. However, the processes, tools, methodologies followed are the same. This aspect of software development shows that the defects in the process are likely to be repeated.

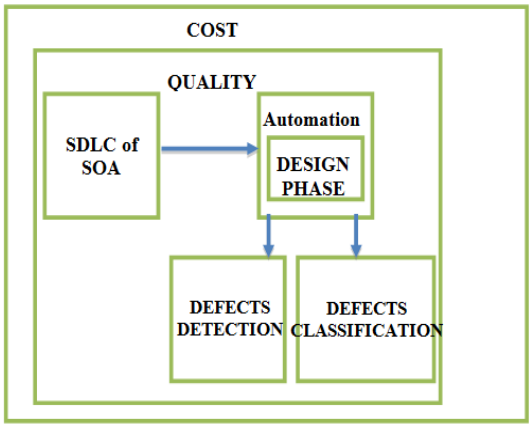
Applying quality management "control" on the software process is being adopted as a guarantee to achieve software quality. Total quality management of the software design aims at continuously improving the quality of the end [2]. Managing the software design by controlling the end product at the design stage is a technique to carve out the causes of defects. This technique adopts a set of practices throughout the software process and is aimed at consistently meeting the end user needs.

The development of code for software development is a practice that requires skill and experience, producing a design defect free code that does not have bugs is a difficult task. There are many tools that assist the programmer with the development of code. These help in the detection and correction of these defects. To effectively perform maintenance, programmers need to accurately detect defects. The classification of these defects would also help formulate guidelines in correcting and avoiding them.

Therefore, this research endeavours to develop, test and validate a framework methodology to be used within an intelligent approach for the purposes of detecting and classifying defects at the design phase of software development life cycle (SDLC) service-oriented architecture (SOA) paradigm, while at the same time balancing the cost and quality of addressing such defects with business needs, functional needs and other considerations that system developers

and designers may need to handle for the domain of the study, as shown in figure 1.

Figure 1: The Domain of the Research



To achieve this goal, the objective of the research is to conceptualize, borrow and customize from known working frameworks, such as object-oriented programming to build a solid framework using automated rule-based intelligent mechanisms to detect and classify defects in software design of SOA. Already, several frameworks have been developed with the aim of improving defect detection and classification [2]. Each framework has been designed in such a way that it can be extended and contextualized to fit into any environment, but with no emphasis on distributed systems and services.

The use of intelligent approach will form the core of the frameworks to define and take advantage of informed and learning frameworks that adapt and extend to various architectures. The intention is not brute force investigation of all available options; rather, an intelligent and guided investigation of the frameworks that define the best combination and projection of defect detection and classification framework.

II. BACKGROUND AND CHALLENGES

Our world runs on software. Every business depends on it, every mobile phone uses it, and even every new car relies on code. Without software, modern civilization would fall apart. So, software quality is an important goal in the software development process. But what exactly is software quality? It's not an easy question to answer, since the concept means different things to different people. One useful way to think about this topic is to divide software quality into two aspects: functional quality and structural quality [3].

The term software architecture intuitively denotes the high level structures of a software system. It can be defined as the set of structures needed to reason about the software system, which comprise the software elements, the relations between them, and the properties of both elements and relations [4]. Systems should be designed with consideration for the user, the system (the IT infrastructure), and the business goals as shown in figure 2.

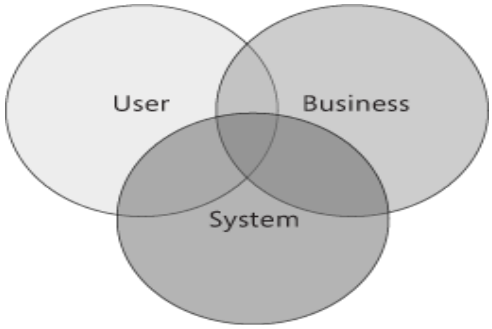


Figure 2: Application Stakeholders [4]

Service-oriented architectures (SOA) are based on the notion of software services, which are high-level software components that include web services. Implementation of an SOA requires tools as well as run-time infrastructure software. One of the most important benefits of SOA is its ease of reuse [5]. But some criticisms of SOA depend on conflating SOA with Web services. For example, some critics claim SOA results in the addition of XML layers, introducing XML parsing and composition. In the absence of native or binary forms of remote procedure call (RPC), applications could run more slowly and require more processing power, increasing costs [6].

SOA provides an evolutionary approach to software development, however, it introduces many distinct concepts and methodologies that need to be defined and explained in order to understand the SOA offerings in an accurate way and build a competent architecture that satisfy the SOA vision. The main issue is to analyze and assess the differences of SOA from past architectural styles, investigate the improvement that SOA has brought to the computing environment, and apply this knowledge to service based application development so as to have a satisfactory SOA.

Software systems have become a crucial part of business and commerce in the modern world. Consequently, software quality has become fundamental in ensuring the proper functioning of the systems and to minimize development and maintenance costs. The quality of software should be guaranteed throughout the entire life cycle of software development, which points toward detecting errors earlier during development.

One obvious and common challenge facing software quality is the detection of design defects and their correction. The main objective of that is to achieve complete customer satisfaction. One of the important steps towards total customer satisfaction is the generation of nearly zero-defect products [7]. The defect management process includes defect prevention, defect discovery and resolution, defect causal analysis, and process improvement [8].

Another challenge involves the application architecture, because it seeks to build a bridge between business requirements and technical requirements by understanding all of the technical and operational requirements, while optimizing common quality attributes. An architecture is the set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behaviour as specified in the collaborations among those elements, the composition of these structural elements and behavioural elements into progressively larger subsystems, and the architecture style that guides this organization — these elements and their interfaces, their collaborations, and their composition [9].

The Application Architecture (AA) describes the layout of an application's deployment. This generally includes partitioned application logic and deployment to application server engines. It relies less on specific tools or language technology than on standardized middleware options, communications protocols, data gateways, and platform infrastructures such as Component Object Model (COM), JavaBeans and Common Object Request Broker Architecture (CORBA).

The application architecture is used as a blueprint to ensure that the underlying modules of an application will support future growth. Growth can come in the areas of future interoperability, increased resource demand, or increased reliability requirements. With a completed architecture, stakeholders understand the complexities of the underlying components should changes be necessary in the future. The application architect is tasked with specifying an AA and supporting the deployment implementation.

Another challenge relates to providing a framework that will improve the defect prevention process. The following aims and objectives will lead to the model of a framework that will improve the defect prevention process:

- Analyze SOA quality and identify the common features of the quality models.
- Analyze the problems of automating the detection and the correction of software design defects.
- Find out the most common quality metrics that can be used to assess the impacts of design defect on software quality.
- Use multi-criteria decision-making tools to analyze QoS quality characteristics in accessing and making decisions on prioritization of design patterns.
- Develop a guideline or framework to automate the detection of design defects based on design patterns and using design constraints.

III. CASE STUDY AND EVALUATION

3.1 Introduction

The design of the framework is only as good as the analysis, and the basic overarching question at this phase is “How will the framework actually work?”. Thus, this section presents the evaluation of the proposed framework, particularly its "Design Defects Measuring Matrix" firstly using research tool based on a questionnaire and workshop in order to assess the different phases of the framework. Secondly, a case study commonly used in service-oriented systems with a number of design approaches is considered in order not only to evaluate the framework but also to check the impact of different architectural styles on both software defects and software quality.

A part of framework evaluation consists of capturing the quality attributes the architecture must handle and to prioritize the control of these attributes. If the list of the quality attributes is suitable in the sense that at least all the business objectives are indirectly considered, then, we can keep working with the same architecture. Otherwise, an alternative architecture that is more suitable for the business should be considered. These quality attributes may be conflictive for achieving business objectives. In such a case, it should be focused on a limited set of attributes, especially if the evaluation of the architecture gives a positive result in a business and a poor one in another one figure 3.

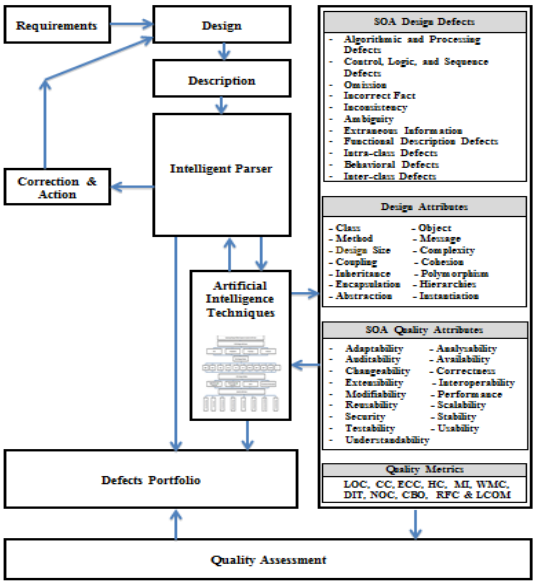


Figure 3 DESQA Framework

3.2 Research Tool

The success of the preliminary evaluation of the framework depends on how well the questionnaire is constructed. In this section, the research tool is a questionnaire. The designed questionnaire examines the relationship between service-oriented architectures (SOAs) and quality attributes.

3.3 Results and Discussion

The main objective of this case study is to demonstrate the usability or practical applicability of the proposed Design Defects Measuring Matrix. The second group required to calculate the expected quality level of the designed software based on the design defects using the following formula:

$$\sum_{i=1}^{n} \sum_{j=1}^{mi} (MI_i \quad X \quad WA_{ij}\ )/mij$$

Defect related to Algorithmic and Processing Defects will affect the following quality attributes.The weights of metrics and quality attributes are used to calculate the total impact on quality.

3.4Case Study: Automated Teller Machine (ATM)

This section demonstrates the applicability and use of the framework proposed in the previous section, and describes how its components are deployed in a case study which is based on a typical banking service namely, the Automated Teller Machine (ATM) [10]. The choice of the case study is driven by the fact that the ATM provides a service that is communicating with a number of banking services such as authentication, transactions, reporting etc. within one bank as well as communication and obtaining services from other banks. Thus, ATM processes require communication between a numbers of components/services to complete a user request, including transaction, client (user interface) and back-end service as well as authentications etc.

ATM, as a case study, has been commonly used in early object-oriented systems [11]. In addition, it exhibits the service concepts reflect in client/server architecture with the banking sector including different modules for front end (user interface) and back engine services. ATM services are relatively easy to model, and can be used as a proof of concept for evaluating the proposed framework. In addition it can be modeled using different designs that broadly follow the SOA principles. Thus, it can be used for testing different SOA architectural styles.

In order to meet comprehensive ATM requirements and system analysis with requirement specification both functional and non-functional requirements need to be considered for the design and development of service-oriented based architectures, that can be used and compared both in terms of potential defects and quality estimation for different SOA architectural styles [12].

3.4.1 Requirements aspects

An ATM provides money to authorised users who have sufficient funds on deposit. It requires the user to provide a personal identification number (PIN) as an authorisation. Money is provided after a confirmation from the bank's computer system. Overall the main function of the ATM is to provide a number of services to the customer:

A customer must be able to make a cash withdrawal from any suitable account linked to his/her card. A customer must be able to make a deposit to any account linked to the card, consisting of cash and/or cheques in an envelope. The customer will enter the amount of the deposit into the ATM, subject to manual verification when the envelope is removed from the machine by an operator.

A customer must be able to make a balance inquiry of any account linked to the card.

A customer must be able to abort a transaction in progress by pressing the Cancel key instead of responding to a request from the machine.

A customer must be able to print the balance, mini-statements, receipts etc.

Transfer money, change PINs etc.

The ATM will service one customer at a time. A customer will be required to insert an ATM card and enter a PIN. The customer will then be able to perform one or more transactions. The card will be retained in the machine until the customer indicates that he/she desires no further transactions, at which point it will be returned.

The ATM will have a key-operated switch that will allow an operator to start and stop the servicing of customers. After turning the switch to the "on" position, the operator will be required to verify and enter the total cash on hand. The machine can only be turned off when it is not servicing a customer. When the switch is moved to the "off" position, the machine will shut down, so that the operator may remove deposit envelopes and reload the machine with cash, blank receipts, etc.

As well as functional requirements there are a number of non-functional requirements i.e. expected quality requirement such as:
Performance — how long does a transaction take?
Availability — what are the hours of operations?
Security — how to identify the client
Usability — is the client able to cancel the operation?
Modifiability — how long does it take to change the authentication mechanism?
Reusability — how easy is it to reuse existing components?

By applying the framework, the potential defects in the application development will be identified thus the number of defects leaking to the implementation stage will be reduced. In addition an estimation of the quality requirements and quality factors will be produced.

3.4.2 Design Aspects

At a high level the ATM machine is based on four main services, Authentication Service for user authentication including card verification, PIN etc.

Transaction Service reflecting the required transactions, withdraw, deposit etc. Storage Service which is used for storing the transactions as well as user details, and Client Service that provides the interface to user of the ATM such as menu (Figure 4).

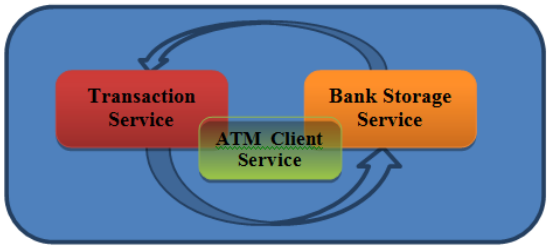


Figure 4: ATM System

Services are linked together for example the Client Service provides an interface on the local machine to invoke other services such as Authentication and Transaction Services. The same applies to other services for example Authentication Service invokes a signal to other services such as Storage Service checking and verifying users. A number of services (use cases) are represented in the use case diagram (Figure 5).

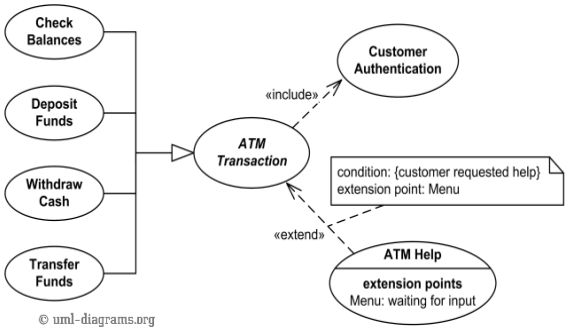


Figure 5 Use Case Diagram [13]

3.4.3 Design Granularity

Having considered both functional and non-functional requirements as well as the main services (use cases) and flow of services, the next stage is to consider the level of granularity of services and its impact on the software quality factors as well as the potential defects. Thus, different designs will be considered, but they should reflect the basic SOA principles, and that can be achieved through a variety of styles/granularity. Thus, in the design of the services and their architecture we seek to evaluate the different granularity i.e. fine grain, coarse grain and thick grain, and their impact on defects as well as quality factors.

They might lead not only to serious defects but even to software failures. Tiny service is an SOA anti-pattern that corresponds to thin service with a small number of methods. This often requires several thin services that are coupled to be used together for the composition of client applications which adds to service management complexity. On the other hand, multi service corresponds to a large service that with a larger number of methods. This might reduce service reusability because of the low cohesion of its methods. Thus, for the ATM case study we apply

the framework using different levels of granularity, fine, coarse and thick. The aim is to produce potential defects portfolio and quality estimation, and providing a comparison between the different granularity levels.

3.4.3.1 Fine Grain

The identified, from user requirements, service candidates are mapped to a typical SOA configuration (Figure 6) and include Authentication, Balance Inquiry, Withdraw, Deposit, etc. Each of the fine grain services were designed to reflect the business logics and rules. In summary all functional requirements and all operations are considered as services i.e. the ATM application is made up of all the individual services. Clearly this is a fine grain approach (tiny services).

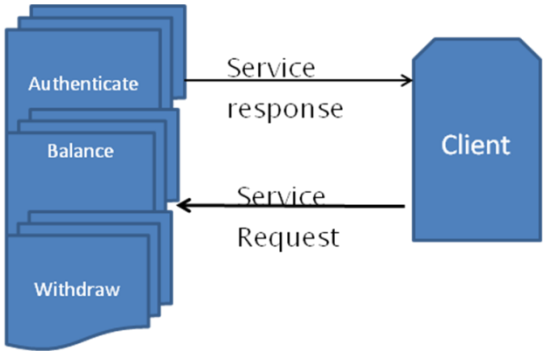


Figure 6 Fine Grain Services

3.4.3.2 Coarse Grain

Next some of the operations discussed in the previous sections are aggregated in a logical and consistent fashion to create the ATM application. The aim is to create coarse grain services that are still meeting all the functional requirements. For, example in Transaction Service will be comprised of a number of operations such as Withdrawal, Deposit, Balance query etc. while Authentication will have check Id, Check PIN, change PIN etc. As shown in Figure 7 the operations are represented by various components/services that follow the principles of SOA.

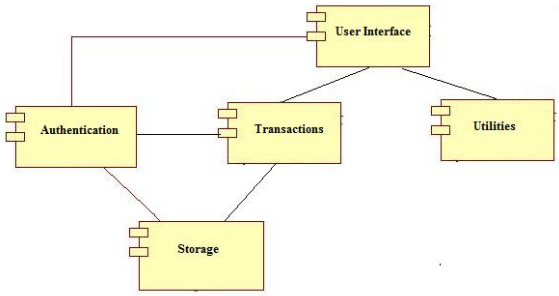


Figure 7 Coarse Grain Services

3.4.3.3 Thick Grain

Finally, some of the services/components presented in the previous section are aggregated from the three tier architecture of the system, with frontend, middleware and backend components, with services/operations mapped to different components still in a logical and consistent fashion to create the ATM application. The aim is to create think grain services that are still meeting all the functional requirements as shown in Figure 8.

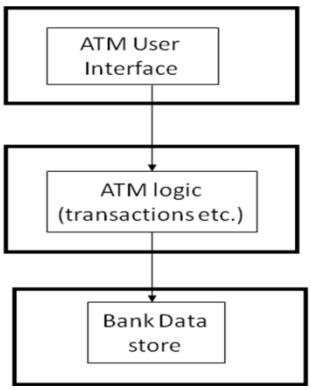


Figure 8 Three Tier Architecture

3.4.4 Evaluation and Observation

The next stage is to apply the framework on the different architectural styles and produce an estimate of defects and the impact of software metrics such as size, complexity, coupling, cohesion etc. and to use the metrics values to produce a quality estimation including the most relevant, from SOA point of view, software quality factors that have been identified in the non-functional requirements. This evaluation will allow us not only to evaluate the framework but also to make a comparison between different SOA based architectural styles.

The first stage is to produce the defects portfolio for the different levels of granularity as shown in figures 9, 10 and 11.

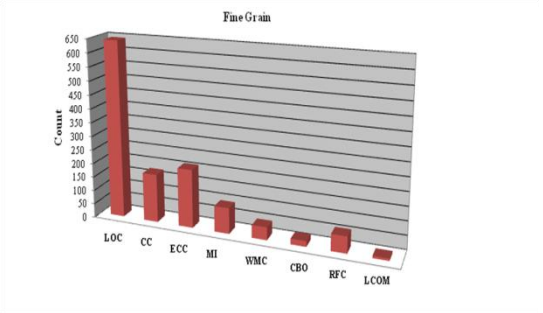


Figure 9 Fine Grain Services

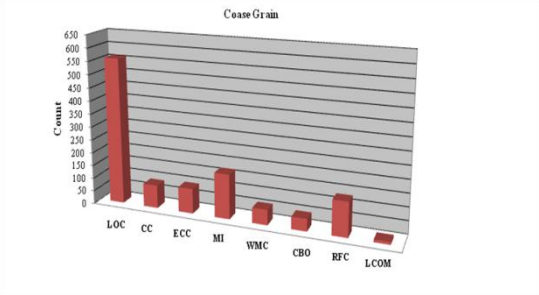


Figure 10 Coarse Grain Services

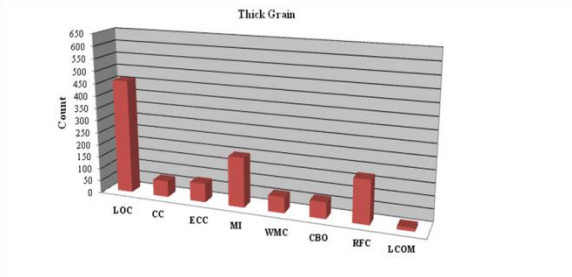


Figure 11 Thick Grain Services

The comparison of the various granularity levels for the case study has shown that fine grain style shows a lower degree of coupling, than the other two styles, coarse and thick grain, in fact the degree of coupling seems to be increasing as we move from fine, to coarse and then to thick services. In terms of complexity thick grain style has the lower complexity followed by coarse grain and finally by fine grain style. The experiment also has shown that the size is highest for fine grain, and the lowest for thick grain due to additional code associated with each layer (Service Interface Layer, Business Layer, and Data Access Layer).

The second phase is to consider the impact on software quality factors, particularly reusability and performance as key factors for SOA applications (Figure 12).

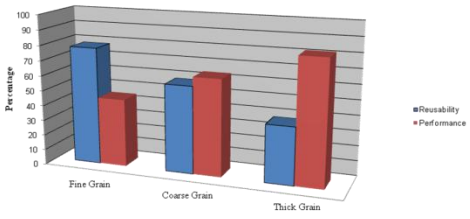


Figure 12 Software Quality Factors

The comparison of the various granularity levels for the case study has shown that fine grain styles tends to promote higher reusability than larger grain styles. In fact the larger the granularity the less reusable individual services become. Performance on the other hand shows the opposite trend, i.e. the higher the granularity the better the performance. This is directly linked to coupling and complexity metrics. Thus, we conclude that there should be a compromise between reusability and performance, so coarse grain services seem to offer this compromise between the two factors (figure 13), but this will at the end depend very much on the type of applications and the user requirements.

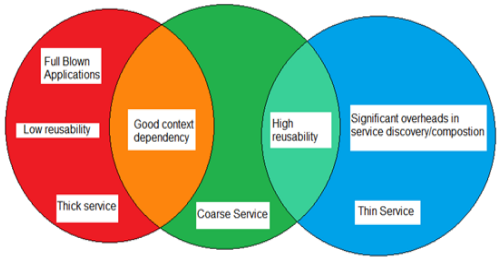


Figure 13 Different Granularity Impacts

Overall, fine grained services are relatively simple and provide small and well specified functionalities. They have the advantage of being easily reusable, i.e. they provide high reusability which is a very important quality factor. They can be used by many services within an application domain or across multiple domains and typically require the transmission of small amounts of data. The disadvantage is that they might become a very large number of services which is hard to manage. This might have negative impact on performance which is another important software quality factor, for example when multiple calls to different services with real time communication and data transfer. On the other hand coarse grained services will be fewer therefore they require less management, with possibly better performance but lower reusability .In addition they might require larger volumes of data to be transmitted and be more complex for other services to use. Thick grain services almost approach full blown applications.

4. Conclusion

Ideally, one would want to optimize for all quality attributes, but the fact is that this is nearly impossible, because any given system has trade-off points that prevent this. Essentially, changing one quality attribute often forces a change in another quality attribute either positively or negatively. The purpose of this section was to investigate the applicability of the DESQA framework and how the design defects measuring matrix can assess attributes of size, complexity, coupling, and cohesion using quality metrics. Thus, after

preliminary research study and date selection to build a defects measuring matrix, a case study was presented where different SOA styles for the same applications were compared using the framework.

However, there are a number of limitations associated with the case study. Firstly, relatively a small numbers of participants were used to design the proposed matrix. Secondly, implementations are not fully operational due to the absence of experiences, although the designs and implementations are structurally complete. Such factors could influence matrix under investigation. In addition, the chosen case study although it is used as a proof of concept it is relatively straightforward, perhaps a larger and more complex case study needs to be considered in future work.

IV. CONCLUSIONS

Quality is an important goal in the software development process and the detection of design defects and their correction early in the development process substantially reduce the cost of subsequent activities of the development and support phases. Bad design and software defects often make source codes hard to understand and lead to maintenance difficulties. Whereas detecting and fixing defects make programs easier to understand by developers. Implementation of corrective and preventive actions is the path towards improvement and effectiveness of software quality. The correction solutions, a combination of refactoring operations, should minimize, as much as possible, the number of defects detected using the detection rules.

Defect prevention practices enhance the ability of software developers to learn from those errors and, more importantly, learn from the mistakes of others. Effective defect tracking begins with a systematic process. It involves a structured problem-solving methodology to identify, analyze and prevent the occurrence of defects. Defect prevention is a framework and ongoing process of collecting the defect data, doing root cause analysis, determining and implementing the corrective actions and sharing the lessons learned to avoid future defects.

Service-oriented architecture (SOA) is an architectural design pattern based on distinct pieces of software providing application functionality to support service-orientation. In this research, a detailed definition and discussion of SOA, its characteristics and principles are presented. The adoption and governance are also discussed. Web services can implement an SOA. So, the web services technology, which is the most appropriate environment to develop SOA currently, is also mentioned. Other technologies for implementing SOA, such as CORBA are also considered.

Software quality measurement is about quantifying to what extent software design possesses desirable characteristics. In this research software quality of service-oriented architecture and its models (McCall quality model, Boehm's quality model, Dromey's generic quality model and ISO quality model) are discussed in detail. The tools of measuring the software quality (quality metrics) are reviewed and discussed.

V. FUTURE WORK

The important limitations of this study are concerned with its generalizability. So, based on the work presented in this thesis, there are a number of areas that can be further improved and carried forward.

The perception of quality differs from individual to individual, a further improvement can be added by redeploying the design defect measuring matrix using large numbers of participants when building it and increasing the number of quality attributes, the number of quality metrics and the number of design attributes. The main purpose of that is to standardize them, to build the trust and the confidence level between the provider and the consumer and will continue to evolve as more new technologies emerge on the horizon.

Although there are areas that could have helped improve the framework significantly, the work presented so far has been able to demonstrate how the aim can be analyzed. The case study discussed in this thesis is limited to one application, the first suggestion is related to the fact that the implementation was carried out in a simulated environment with the results presented. It would be of great benefit for this to be tested in a real-life case. It also will be interesting if the scope is expanded to includelarge number of applications which form part of the whole business. The DESQA framework can be adapted in the design process using its measuring matrix components and can incorporate metrics to measure design defects. A reverse engineering methodology can be added to this system to improve the traceability of individual components of the system or incorporate changes easily. To improve granularity a refined pattern can be added to the future expansion.

The second relates to the extension of framework applications. As seen in the evaluation of the results, the framework was designed with the possibility to extend it to adapt additional quality metrics. The extension can be considered in future work by building complexities to adapt more different design attributes and quality metrics. The last suggestion relates to testing the framework for the impact of larger and potentially conflicting

quality requirements in non-controlled environments.

BIBLIOGRAPHY

[1]. Moha, Gueheneuc& Leduc (2009). Bad Smell in Design Patterns.*Journal of the Object Oriented Technology*.

[2]. Kessentini, M.,Sahraoui, H.,&Boukadoum, M. (2008). Model Transformation as an Optimization Problem.*Proc.MODELS*: 159-173 Vol. 5301 of LNCS. Springer.

[3]. Pressman, S. (2005). Software Engineering: A Practitioner's Approach (Sixth International ed.). McGraw-Hill Education. Pp. 388.

[4]. Clements, P., Bachmann, F., Bass, L.,Garlan, D.,Ivers, J., Little, R.,Merson, P., Nord, R.,&Stafford, J. (2010). Documenting Software Architectures: Views and Beyond, Second Edition. Addison-Wesley,Boston.

[5]. Bell, M. (2008). Introduction to Service-Oriented Modeling. *Service-Oriented Modeling: Service Analysis, Design, and Architecture*. Wiley & Sons.

[6]. M. Riad, Alaa, E. Hassan, Ahmed, & F. Hassan, Qusay (2009). Investigating Performance of XML Web Services in Real-Time Business Systems. *Journal of Computer Science & Systems Biology* 02 (05): 266–271.

[7]. Gopalakrishnan Nair, T.R.,& Suma, V. (2010). The Pattern of Software Defects Spanning across Size Complexity. International Journal of Software Engineering.

[8]. Jäntti, M.,Toroi, T.,&Eerola, A. (2006). Difficulties in Establishing a Defect Management Process: A Case Study. Journal of Software Engineering.Springer.

[9]. Booch, G., Rumbaugh, J.,& Jacobson,I. (1999). The Unified Modeling Language User Guide. Addison-Wesley.Reading, MA.

[10]. Frost, R., Hafiz, R. & Callaghan, P. (2007). Modular and Efficient Top-Down Parsing for Ambiguous Left-Recursive Grammars., *10th International Workshop on Parsing Technologies (IWPT), ACL-SIGPARSEJune 2007, Prague*. Pp 109-120.

[11]. Yingxu, W., Yanan, Z., Philip, C., Xuhui, L. & Hong, G., (2010). The Formal Design Model of an Automatic Teller Machine (ATM). *International Journal of Software Science and Computational Intelligence*, 2(1): 102-131.

[12]. RajniPamnani, PramilaChawan, Satish Salunkhe. Object Oriented UML Modeling for ATM Systems. Department of computer technology, VJTI University, Mumbai.

[13]. Wikipedia. "ATM System" www.wikipedia.org/wiki/Automated_teller_machine