A Framework for the Classification and Detection of Design Defects and Software Quality Assurance

Khaled Kh. S. Kh. Allanqawi

A thesis submitted in partial fulfillment of the requirements of Kingston University for the degree of Doctor of Philosophy

Faculty of Science, Engineering and Computing
Kingston University

February 2015

ACKNOWLEDGEMENTS

Initially, I would like to express my deep sincere gratitude for the invaluable advices, continuous encouragement and scientific support of Prof. **Dr. Souheil Khaddaj** during the course of this research.

Next, I would like to thank Kuwait Government for the financial support, without this support, I could not spend my time researching and trying to do my best to complete this dissertation.

Finally, I would like to thank my great family, my wife, my wonderful kids and my parents. Especially, my parents that always stood by me with everything I needed during my life.

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 framework on design defects and software quality assurance (DESQA) will blend various design defect metrics and quality measurement approaches and will provide measurements for both defect and quality factors. Unlike existing frameworks, mechanisms are incorporated for the conversion of defect metrics into software quality measurements. The framework is evaluated using a research tool supported by sample used to complete the Design Defects Measuring Matrix, and data collection process. In addition, the evaluation using a case study aims to demonstrate the use of the framework on a number of designs and produces an overall picture regarding defects and quality.

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 fulfil 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.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AHF	Attribute Hiding Factor
AIF	Attribute Inheritance Factor
ASOM	Average Services Operation Complexity
AW	Quality Attribute Weight
CAM	Cohesion Among Methods of Class
CBD	Component Based Development
CBO	Coupling Between Object Classes
CC	Cyclomatic Complexity
COF	Coupling Object Factor
DÉCOR	Defect Detection for Correction
DIT	Depth of Inheritance Tree
DIT	Depth of Inheritance Tree
ECC	Error Correcting Code
GUI	Graphical user interface
HC	Halstead's Complexity
ISO	International Organization for Standardization
IT	Information Technology
JVM	Java Virtual Machine
LCOM	Lack of Cohesion of Methods
LOC	Lines Of Code metric
MHF	Method Hiding Factor
MI	Maintainability Index
MIF	Method Inheritance Factor
MI_i	Metric Rank
NIST	National Institute of Standard Technology
NMI	Number of Methods Inherited
NOC	Number Of Children
ODC	Orthogonal Defect Classification
OOD	Object Oriented Development
PF	Polymorphism Factor

POA	Process Oriented Architecture
QA	Quality Attributes
QoS	Quality of Service (QoS)
REF	Response Set for a Class
SDLC	Software Development Life Cycle
SLOC	Source Line of Code
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SOG	Services Operation Granularity
UDDI	Universal Description, Discovery, and Integration
UML	Unified Modeling Language
WMC	Weighted Methods per Class
WSDL	Web Service Description Language

LIST OF EQUATIONS

Eq. No.	Equation
3.1	$COF = \frac{\sum_{i=1}^{TC}\left[\sum_{j=1}^{TC} is\_client(C_i, C_j)\right]}{TC^2 - TC}$
3.2	$WMC\ (c) = \sum_{m \in M_{\text{Im}}(c)} VG\ (m)$
3.3	$ASOM = \frac{\left(\sum_{i=1}^{n}(SOG(i))^2\right)}{NS}$
3.4	$LCOM\ (C) = \frac{NOM - \sum_{a \in C} NOMAF}{NOM - 1}$
3.5	$MI\ F = \frac{\sum_{i=1}^{TC} Mi(Ci)}{\sum_{i=1}^{TC} Ma(Ci)}$
3.6	$AIF = \frac{\sum_{i=1}^{TC} Ai(Ci)}{\sum_{i=1}^{TC} Aa(Ci)}$
3.7	$POF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC}\left[M_n(C_i) \times DC(C_i)\right]}$
3.8	$MHF = \frac{\sum_{i=1}^{TC}\sum_{m=1}^{Md(Ci)}(1 - V(Mmi))}{\sum_{i=1}^{TC} Md(Ci)}$

3.9	$AHF = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{Ad(Ci)} (1 - V(Ami))}{\sum_{i=1}^{TC} Ad(Ci)}$
5.1	$\sum_{i=1}^{n} \sum_{j=1}^{mi} (MI_i \quad X \quad WA_{ij} \ )/mij$

CHAPTER 1: OVERVIEW AND BACKGROUND

1.1 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.

While focusing on the software design defects, it is important to note that poor customer requirements elicitation could contribute to poor design of the software [1]. The focus here is the practices of software management adopted to counter software defects and detect the defects. Most importantly, the main idea is using established processes to catch the software design defects. From this perspective, we are able to examine how total quality management and continuous management of the process are affected using the design.

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 1.1.

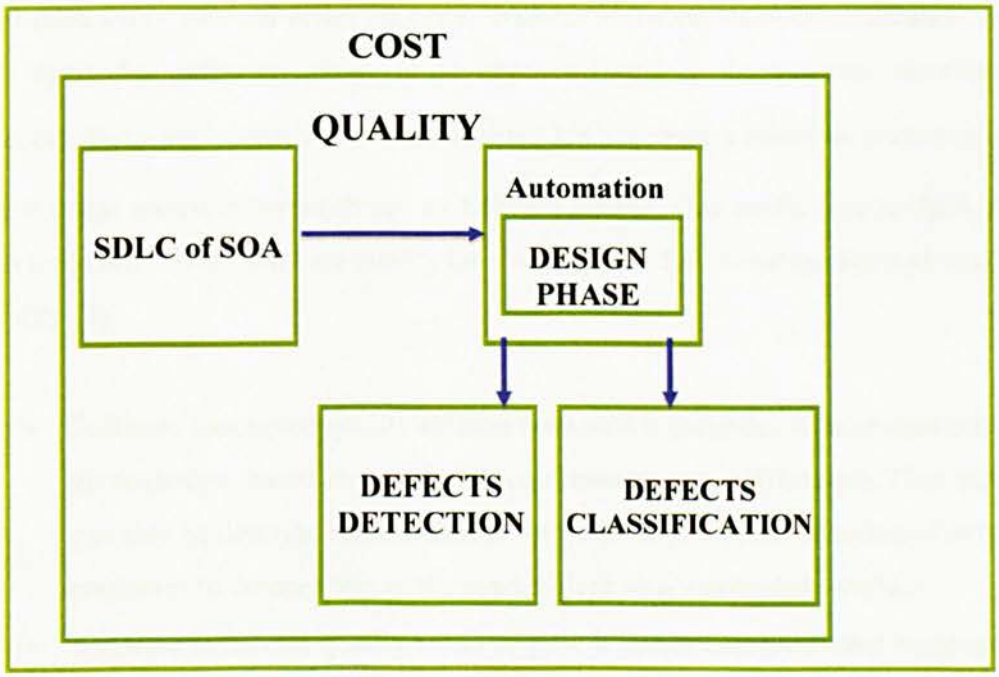

Figure 1.1: The Domain of the Research

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.

This chapter gives the background and challenges of studying software especially in the design phase. The last section describes the structure of the thesis.

1.2 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].

- Software functional quality reflects how well it complies with or conforms to a given design, based on functional requirements or specifications. That attribute can also be described as the fitness for purpose of a piece of software or how it compares to competitors in the marketplace as a worthwhile product.
- Software structural quality refers to how it meets non-functional requirements that support the delivery of the functional requirements, such as robustness or maintainability, the degree to which the software was produced correctly.

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 1.2.

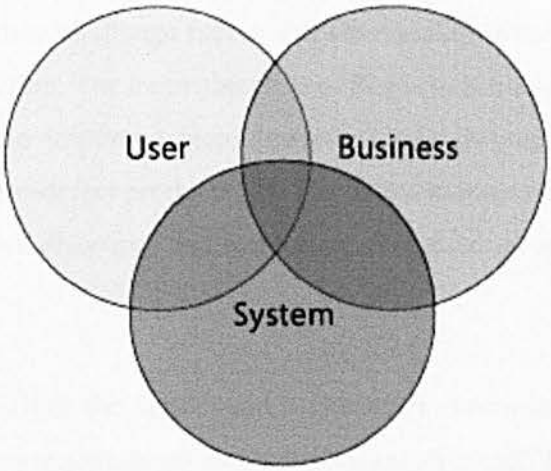

Figure 1.2: Application Stakeholders [4]

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.

1.3 Aims and Objectives

Defect prevention is an important activity in any software project. Consequently, the study aimed at:

- Investigating the service-oriented architecture (SOA) concept and its applications and design defects.
- Investigating the enhancement of software quality.
- Investigating the most common design defects in the software industry.
- Analyzing the most important and key activities in the system development life cycle (SDLC) phase that ensures the quality of software.
- Developing a new framework "design defects and software quality assurance framework (DESQA)" for defect tracking and detection and quality estimation for early stages particularly for the design stage of the SDLC.

1.4 Methodology and Framework Construction

1.4.1 Methodology

The Scope of Work for this study was divided into the following tasks:

- Review of Existing Work and Collection of Supplemental Information
- Propose a framework to automate the detection of design defects:
  - Framework Assumptions
  - Framework Description
  - Framework Formalization
  - Framework Design Execution
- Describe the evaluation of the proposed framework:
  - Research Tool
  - Sample Used
  - Data Collection
- Discuss the results

Finally, find out conclusions and set up future work.

1.4.2 The Framework

The DESQA framework has a number of core components:

- The first component is the **requirements,** which had been set by the client and analysed and validated by the business analysis team.
- The second component of the framework is the **design,** in which the design team set the blueprint of the system, which consists of all major activities and functions of the software.
- The third component is the **description,** when the design is converted to a set of textual forms.
- The fourth component — the **intelligent parser** — considers the main part of the framework. In this component, a deep investigation goes through all parts of the design description and examines the functionality and the systematic design.
- The fifth component — **the defect database** — interacts with the fourth component which contains all the defects that had been classified according to their type.

- The sixth is the "**defects portfolio**", a type of report showing the defects and their classifications. The final component is the correction and action, which is based on the results of the processing mechanism of the intelligent parser.
- The seventh is the **"Quality Assessment** "which links the defects and their metric measurements with software quality factors to produce an estimation of software quality from the design stage.

The framework addresses the problem of defects and quality assurance in the early stages of SDLC for service-oriented architecture and proposes a novel framework to address the identified problems.

1.5 Structure of Thesis

The thesis follows the structure depicted in figure 1.3 starting with the introduction and problem setting all the way to conclusions and future work.

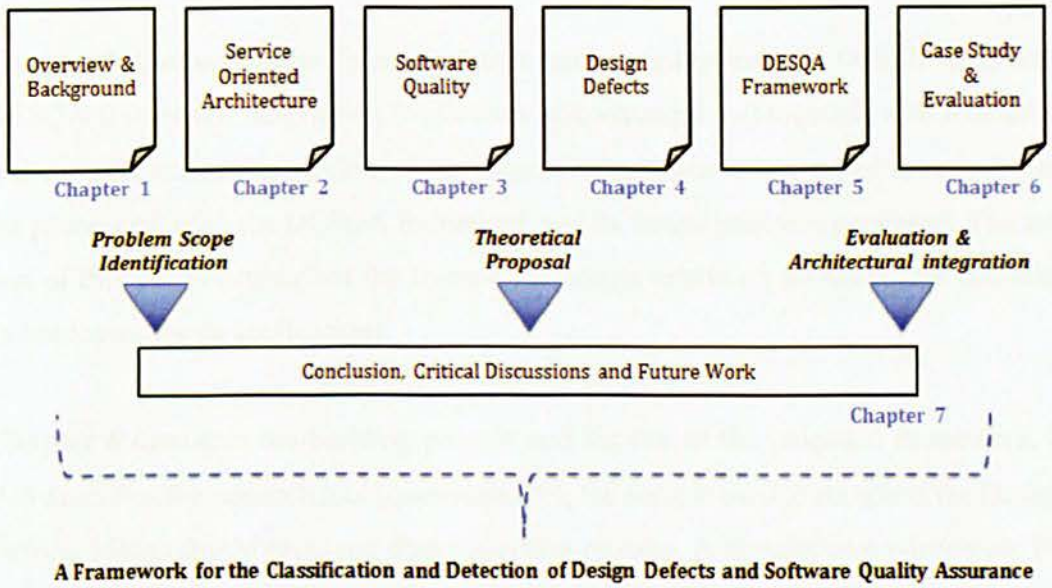

Figure 1.3: Abstract View of Key Aim and Approach of Each Chapter in the Thesis

Chapter 2 is an introduction to service-oriented architecture definition, characteristics, principles, adoption and governance. 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. Also in this chapter; service oriented architecture strategies are discussed.

Chapter 3 introduces software quality of service-oriented architecture and discusses its models (McCall quality model, Boehm's quality model, Dromey's generic quality model and ISO quality model). The last part of this chapter describes Quality Metrics of Service Oriented Architecture and reviewing the different metrics of SOA. It also focuses on how to measure the quality metrics and their impacts on SOA quality.

Chapter 4 covers the definition of defects in system development life cycle including *defects classification* and defects categories, followed by a detailed description of design defects. Design attributes: class, object, method, message instantiation, inheritance, polymorphism, encapsulation, cohesion, coupling, design size, hierarchies, abstraction and complexity are defined. Also in this chapter; defect detection categories and strategies are discussed.

Chapter 5 discusses a new framework that can be used to measure QoS. It starts with DESQA framework description (objectives and assumptions) together with a detailed and comprehensive description of the proposed framework. A detailed description of the process of using the DESQA framework and its formalization is presented. The last part of this chapter describes the framework design execution including the potential technologies for its applications.

Chapter 6 discusses the building process and the use of the proposed framework. It also describes the research tool (questionnaire), the sample used to complete the Design Defects Measuring Matrix, and data collection process. A simulation environment for the DESQA framework applications is presented using a case study. Finally, the results of the simulation and conclusion are presented.

Finally, **Chapter 7** concludes the thesis by highlighting the main contribution in service-oriented architecture. It gives a summary of the contribution and work done as well as the method of evaluation and results. It also highlights further areas of research that can be carried out in the areas of SOA.

CHAPTER 2: SERVICE-ORIENTED ARCHITECTURE

2.1 Introduction

It is difficult to define what a service-oriented architecture (SOA) is. The term is being used in an increasing number of contexts with conflicting understandings of implicit terminology and components. SOA is a collection of independent loosely coupled applications that are capable of communicating in the form of provision of service (e.g. data transmission) [10]. SOA is defined in many literatures with extensive number of articles attempting to define what it means and how it can be used and implemented in an organization. In [10], [11] SOA is defined by identifying a number of specific characteristics such as loose coupling, flexibility, connection and communication among others to encompass and differentiate between a modular function and that of a service function. Some of these characteristics can be associated with the more traditional enterprise architectures for multi-tier application development that provided the foundation from which SOA evolved [12].

CORBA, J2EE and other middleware platforms provided the gluing technology for such enterprise applications separating the independence of the application from the implementation technology, thus making it easier for organizations to deploy independent applications readily when needed. Traditional, enterprise applications based on the like J2EE and CORBA were constructed using component based development (CBD) or object oriented development (OOD) encapsulating the business modules in the form of components or objects offering specific solutions to the business.

Services in SOA conform to a standard format to enable easy accessibility and communication, and independence of the underlying development technology. Services represent the blocks that are required in SOA; they are the individual components that collectively define the SOA. SOA is not a definite framework of products that can be purchased and implemented, however it represents the technical design framework of how to develop applications as services within an organization. This chapter starts with SOA definition and adoption, followed by web services in section 2.4. SOA strategies

are also discussed in section 2.5. Finally, a summary of the chapter is presented in section 2.6.

2.2 Service-Oriented Architecture

There is no single, official definition of what an SOA is. Consequently, many of the organizations promoting the use of SOAs and building technologies to make it easier for organizations to adopt an SOA approach have defined the term. As a result, SOA is defined in many different ways as follows:

- SOA is an application framework that takes everyday business applications and breaks them down into individual business functions and processes, called services. An SOA lets you build, deploy and integrate these services independent of applications and the computing platforms on which they run [13].
- SOA is an approach to organizing information technology in which data, logic, and infrastructure resources are accessed by routing messages between network interfaces [14].
- An SOA is a set of components which can be invoked, and whose interface descriptions can be published and discovered [15].
- SOA is an architectural style that supports service-orientation. SOA is a software design and software architecture design pattern based on structured collections of discrete software modules, known as services that collectively provide the complete functionality of a large software application [16].
- SOA is a paradigm for developing distributed systems that deliver application functionality as a set of services, which are reused by end-user applications and other coarse-grained services [17].

Josuttis defined SOA as a collection of independent loosely coupled applications that are capable of communicating in the form of provision of service (e.g. data transmission) [10]. Sprott and Wilkes [18] defined SOA as: The policies, practices, frameworks that enable application functionality to be provided and consumed as sets of services published at a granularity relevant to the service consumer. Services can be

invoked, published and discovered, and are abstracted away from the implementation using a single, standards-based form of interface. In 2012, Erl et al. [19] redefined SOA as: A technology architectural pattern for service oriented solutions with distinct characteristics in support of realizing service orientation and the strategic goals associated with service-oriented computing.

The purpose of SOA is to allow easy cooperation of a large number of computers that are connected over a network. Every computer can run an arbitrary number of programs — called services in this context — that are built in a way that they can exchange information with any other service within the reach of the network without human interaction and without the need to make changes to the underlying program itself.

Services can be categorized into fine-grained (less functionality or operations), coarse-grained or thick-grained (more functionality or operations). Within the concept of SOA the more functionality with less functional operations can improve the execution process. SOA brings together the interoperation and integration of technologies in business processes within a framework of the enterprise consumer. The technologies implemented and deployed for example using web services offer a layer of system integration where software is used to implement a business process as a service which can be made of functions or other services. With internet technology and web services, SOA services are capable of been utilised beyond the domain of origin in the form of added service or functionality delivered to consumers needing such specific services. However, these require many of the distributed computing quality attributes in effective implementation of delivery, interoperability, security, performance, availability, scalability among others to meet the requirement of consumers.

In order to measure the need of services and their effective distribution, decomposing services based on their functional and operational attribute offers the chance to be able to incorporate the accessibility of services in a distributed environment such as Cloud Computing. SOA services are independent and loosely coupled to offer great flexibility with services, not hierarchical inherent of other services. However, the purpose of decomposition will show the close matching of services in the form of execution to

represent the hierarchical, sequential and parallel layers of functions with dependencies of functions in the service.

Figure 2.1 shows the architectural implementation of SOA, showing technologies like UDDI, WSDL and web-services for the listing and invocation of requests. However, the acceptance of Simple Object Access Protocol (SOAP) by World Wide Web Consortium (W3C), HTTP has provided the framework for continuous development of services and for inter-service communication. While XML provided the format for transmission of data and information using SOAP that is platform independent and non-domain specific for any technologies. The ability to offer applications to vendors without the complication of a particular platform or technology using standard and simple HTTP communication with XML has contributed to the domination of SOA and its acceptance as an architectural style [14].

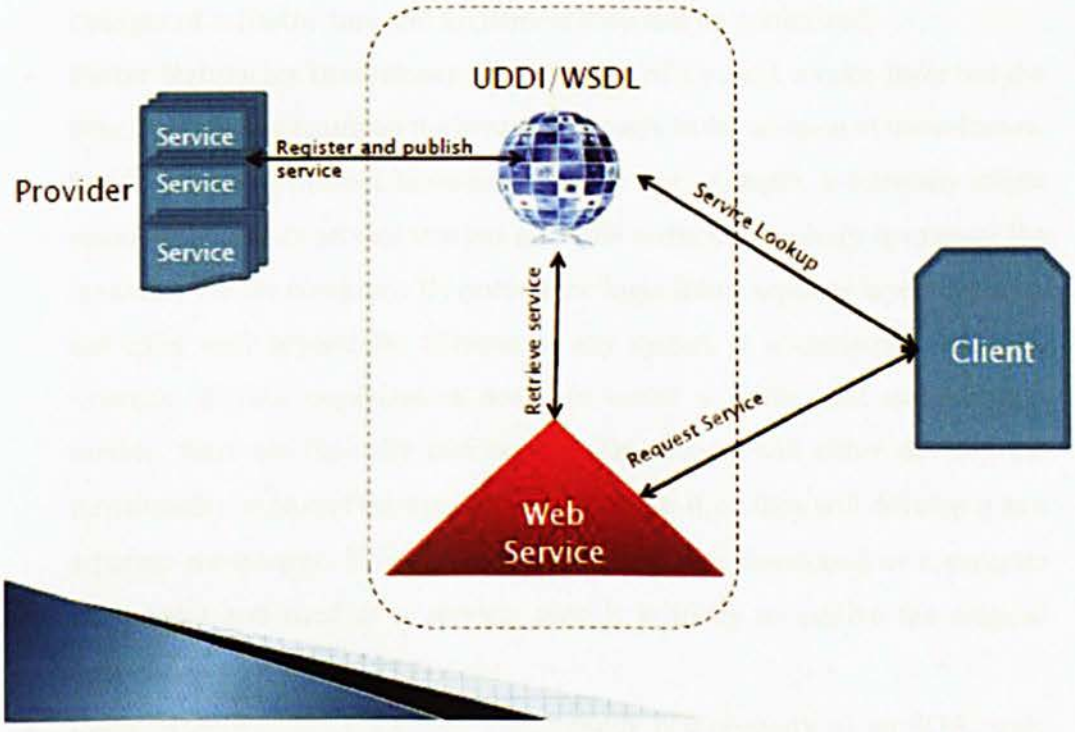

Figure 2.1: Overview of SOA Architecture [14]

Advantages of SOA: The concept of SOA is widely accepted as a software architecture design paradigm which promises the design and implementation of flexible systems and facilitates the change of business processes quickly. SOA leverages the alignment

of business processes and information technology (IT). SOA can be used to combine business services and IT resources. Based on components, SOA can transfer business processes into a set of services linked to each other. Service consumers can use or access these services through a network when necessary. SOA aligns the IT resource with business processes by changing its IT architecture, and this brings many benefits as follows [11, 17 & 20]:

- Maintaining the consistency of IT and business makes it easy to construct a reusable business application system with flexible structure.
- SOA provides an abstract layer through which enterprises can keep on using its IT investment, so as to get maximum utilization of IT assets.
- In SOA, systems are constructed by orchestrating different services which is loosely coupled, platform independent and access transparent. This makes systems much easier to integrate, manage and evolve, and impact on the changes of infrastructure and implementation can be minimized.
- **Better Return on Investment:** The creation of a robust service layer has the benefit of a better return on the investment made in the creation of the software. Services map to distinct business domains. For example, a company might create an inventory service that has all of the methods necessary to manage the inventory for the company. By putting the logic into a separate layer, the layer can exist well beyond the lifetime of any system it is composed into. For example, if your organization needs to create a credit card authorization service, there are basically two options. Developers will either develop the functionality as part of the application that needs it, or they will develop it as a separate component. If credit card authorization is developed as a separate component and used as a service, then it is likely to outlive the original application.
- **Code Mobility:** Since location transparency is a property of an SOA, code mobility becomes a reality. The lookup and dynamic binding to a service means that the client does not care where the service is located. Therefore, an organization has the flexibility to move services to different machines, or to move a service to an external provider.
- **Focused Developer Roles:** An SOA will force an application to have multiple layers. Each layer has a set of specific roles for developers. For instance, the

service layer needs developers that have experience in data formats, business logic, persistence mechanisms, transaction control, etc. A client developer needs to know technologies such as SWING, JSP or MFC. Each layer requires a complex set of skills. To the extent that developers can specialize, they will excel at their craft in a particular layer of the application. Companies will also benefit by not having to rely on highly experienced generalists for application development. They may use less experienced developers for focused development efforts.

- **Better Testing/Fewer Defects:** Services have published interfaces that can be tested easily by developers by writing unit tests. Developers can use tools such as JUnit for creating test suites. These test suites can be run to validate the service independently from any application that uses the service. It is also a good practice to run the unit tests during an automated build process. There is no reason for a QA tester to test an application if the unit tests did not complete successfully. More and better testing usually means fewer defects and a higher overall level of quality.
- **Support for Multiple Client Types:** As a benefit of an SOA, companies may use multiple clients and multiple client types to access a service. A PDA using J2ME may access a service via HTTP, and a SWING client may access the same service via RMI. Since the layers are split into client and service layers, different client types are easier to implement.
- **Service Assembly:** The services that developers create will evolve into a catalog of reusable services. Customers will come to understand this catalog as a set of reusable assets that can be combined in ways not conceived by their originators. Everyone will benefit from new applications being developed more quickly as a result of this catalog of reusable services.
- **Better Maintainability:** Software archaeology is the task of locating and correcting defects in code. By focusing on the service layer as the location for your business logic, maintainability increases because developers can more easily locate and correct defects.
- **More Reuse:** Code reuse has been the most talked about form of reuse over the last four decades of software development. Unfortunately, it is hard to achieve due to language and platform incompatibilies. Component or service reuse is much easier to achieve. Run-time service reuse is as easy as finding a service in

the directory, and binding to it. The developer does not have to worry about compiler versions, platforms, and other incompatibilities that make code reuse difficult.

- **Better Parallelism in Development:** The benefit of multiple layers means that multiple developers can work on those layers independently. Developers should create interface contracts at the start of a project and be able to create their parts independently of one another.
- **Better Scalability:** One of the requirements of an SOA is location transparency. To achieve location transparency, applications look up services in a directory and bind to them dynamically at run-time. This feature promotes scalability since a load-balancer may forward requests to multiple service instances without the knowledge of the service client.
- **Higher Availability:** Also because of location transparency, multiple servers may have multiple instances of a service running on them. If a network segment or a machine goes down, a dispatcher can redirect requests to another service without the client's knowledge.
- **Efficiency:** The ability to quickly and easily create new services and new applications using a combination of new and old services, along with the ability to focus on the data to be shared rather than the implementation underneath.
- **Loose technology coupling:** The ability to model services independently of their execution environment and create messages that can be sent to any service.
- **Division of responsibility:** The ability to more easily allow business people to concentrate on business issues, technical people to concentrate on technology issues, and for both groups to collaborate using the service contract.

SOA framework: SOA-based solutions endeavour to enable business objectives while building an enterprise-quality system. SOA architecture is viewed as five horizontal layers [21]:

1. **Consumer Interface Layer:** These are GUI for end users or apps accessing apps/service interfaces.
2. **Business Process Layer:** These are choreographed services representing business-use cases in terms of applications.

3. **Services:** Services are consolidated together for a whole enterprise in a service inventory.
4. **Service Components:** The components used to build the services, like functional and technical libraries, technological interfaces etc.
5. **Operational Systems:** This layer contains the data models, enterprise data repository, technological platforms etc.

There are four cross-cutting vertical layers, each of which are applied to and supported by each of the horizontal layers [21]:

1. **Integration Layer:** Starts with platform integration (protocols support), data integration, service integration, application integration, leading to enterprise application integration supporting B2B and B2C.
2. **Quality of Service:** Security, availability, performance etc. constitute the quality of service which is configured based on required SLAs, OLAs.
3. **Informational:** Provide business information.
4. **Governance:** IT strategy is governed to each horizontal layer to achieve required operating and capability model.

SOA vision: The SOA vision statement should describe what is to be accomplished with the SOA implementation and in a high-level overview how the organization plans to achieve its goals. Optimally, the SOA vision statement would be the result of activities performed before defining a governance model, though it can be created as part of the SGMM engagement. In addition to the vision, the organization ideally will create a strategy and roadmap statement before the engagement begins [22].

2.3 Service-Oriented Architecture Characteristics

SOA is not appropriated for all types of systems. However, SOA copes well with many difficult-to-handle characteristics of large systems as described next [19, 23 & 24]:

Distributed Systems

SOA systems are normally large and distributed. When business grows, it becomes more and more complex, and more organizations get involved in it. In this context,

SOA solutions support integration of systems from different companies that may be developed in different platforms and programming languages.

SOA is a software architectural concept where applications are partitioned into discrete units of functionality called 'services'. Each service implements a small set of related business rules or function points. These services are made available to consumers/client applications. Whenever a business rule must be modified to support changing business requirements, only the service which implements that business rule needs modification, while the remainder of the application remains intact.
An SOA service can join (registering a profile) or leave (unregistering the profile) the system anytime; e.g., peer to peer applications. To this aim SOAs are operated by so-called Enabling Services, which are in charge of keeping the map of available services and offer *discovery* mechanisms. Typically, services are designed to discover on demand and interact with services of given types (i.e. providing certain APIs and relative functionalities), rather than to refer to known instances of services. The figure below shows how the search scenario above can be interpreted in a SOA.

Different Owners

Another characteristic of SOA systems is that beside large and distributed, different parts of the SOA solution may be under control of different ownership domains. The presence of systems controlled by different owners is the key for certain properties of SOA, and the major reason why an SOA is not only a technical concept.

Heterogeneity

Large systems differ from small systems due to their lack of harmony. Large systems use different platforms and programming languages as already mentioned. They may be composed by mainframes, databases, Java applications, and so on. In other words, they are heterogeneous.

User Access and Security

One of the ways in which SOA can empower a workforce is the creation of a single point sign on. The SOA solution should offer a browser based role-oriented experience for the user which incorporates task lists based on the user's roles and the relevant

collaboration and knowledge content as well as links to the key web sites for the role. The most critical parts of this are the concepts of enterprise-wide identity management and federated identity management (across enterprises) which allow the user to sign on once and for the appropriate access to be given in any application/task the user can access. Given that an SOA environment inevitably will communicate both across the internet and intranet, the set-up of appropriate firewalls to control external (internet) access is a critical factor as for any system which needs to allow external access.

Workflow

Availability of workflow functionality in any SOA solution facilitates the following:

- Easy linking of processes / process parts.
- Linking into the underlying applications where necessary (it is a utopian concept to imagine that ALL processes, across the whole enterprise, will be abstracted into web services some processes may remain within the application).
- Browser-based task lists for the users.

Resilience

As for any other IT architecture, an SOA must provide sufficient resilience to support the business. The SOA run-time environment must facilitate this via enterprise-wide alert handling and, if possible, the ability to target process flows on a specific server (or server group) as a fallback (prioritization of flows).

2.4 Service-Oriented Architecture Principles

SOA is based on a set of service-oriented principles that support its theories and characteristics. The set of principles that are directly related to SOA are described next [10, 19, 20 & 23]:

Coupling: It refers to the number of dependencies among services. In SOA, services maintain a relationship that minimizes dependencies and only requires that services retain an awareness of each other. Loose coupling is achieved

through the use of standards and service contracts among consumers and providers that allow services to interact through well-defined interfaces. Coupling also affects other quality attributes, e.g., modifiability and reusability.

Service contract: Services adhere to communication agreements, as defined collectively by one or more service descriptions and related documents. They define data formats, rules and characteristics of the services and their operations. These documents are defined using standards in order to be readable by the software elements of the architecture, e.g., XML, WSDL and XSD policy documents.

Autonomy and Abstraction: Services have control over the logic they encapsulate, i.e., they must be autonomous and self-contained. Moreover, beyond what is described in the service contract, services hide internal logic from the outside world. Services are like black boxes, and service consumers only depend on the provided interface.

Stateless and Idempotent: Services minimize retaining information specific to a customer's request. They should be as more stateless as possible in order to increase reusability and scalability. Moreover, services should also be idempotent, which is the ability to redo an operation without causing problems, e.g., duplicated data.

Discoverability and Dynamic Binding: Services are designed to be apparently descriptive so that they can be found and accessed via available discovery mechanisms, e.g., Universal Description, Discovery, and Integration (UDDI). The use of a directory is not obligatory but a service should be discoverable. Service discoverability is usually achieved through a third-party entity that implements a discovery mechanism, e.g., a service registry.

Coarse-Grained Interfaces: Services are abstractions that support the separation of concerns and information hiding. However, they slow down performance due to the remote calls. For this reason, services should provide coarse-grained operations that transfer all the necessary data all together instead

of having several fine-grained calls. The requirements of the service consumers should be taken into account when deciding the right granularity for the whole services as well as their operations in order to avoid unnecessary data transfers and performance problems.

Explicit Boundaries: Everything needed by the service to provide its functionality should be passed to it when it is invoked. All access to the service should be via its publicly exposed interface; no hidden assumptions must be necessary to invoke the service. Services are inextricably tied to messaging in that the only way into and out of a service are through messages. A service invocation should as a general pattern not rely on a shared context; instead service invocations should be modeled as stateless. An interface exposed by a service is governed by a contract that describes its functional and non-functional capabilities and characteristics. The invocation of a service is an action that has a business effect, is possibly expensive in terms of resource consumption, and introduces a category of errors different from those of a local method invocation or remote procedure call. A service invocation is not a remote procedure call.

Shared Contract and Schema, not Class: Starting from a service description (a contract), both a service consumer and a service provider should have everything they need to consume or provide the service. Following the principle of loose coupling, a service provider cannot rely on the consumer's ability to reuse any code that it provides in its own environment; after all, it might be using a different development or runtime environment. This principle puts severe limits on the type of data that can be exchanged in an SOA. Ideally, the data is exchanged as XML documents validatable against one or more schemas, since these are supported in every programming environment one can imagine.

Policy-driven: To interact with a service, two orthogonal requirement sets have to be met:

1. The functionality, syntax and semantics of the provider must fit the consumer's requirements.
2. The technical capabilities and needs must match.

To support access to a service from the largest possible number of differently equipped and capable consumers, a policy mechanism has been introduced as part of the SOA tool set. While the functional aspects are described in the service interface, the orthogonal, non-functional capabilities and needs are specified using policies.

Autonomous: Related to the explicit boundaries principle, a service is autonomous in that its only relation to the outside world at least from the SOA perspective is through its interface. In particular, it must be possible to change a service's runtime environment, e.g. from a lightweight prototype implementation to a full-blown, application server-based collection of collaborating components, without any effect on its consumers. Services can be changed and deployed, versioned and managed independently of each other. A service provider cannot rely on the ability of its consumers to quickly adapt to a new version of the service; some of them might not even be able or willing to adapt to a new version of a service interface at all (especially if they are outside the service provider's sphere of control).

Wire formats, not Programming Language APIs: Services are exposed using a specific wire format that needs to be supported. This principle is strongly related to the first two principles, but introduces a new perspective: To ensure the utmost accessibility (and therefore, long-term usability), a service must be accessible from any platform that supports the exchange of messages adhering to the service interface as long as the interaction conforms to the policy defined for the service.

Document-oriented: To interact with services, data is passed as documents. A document is an explicitly modeled, hierarchical container for data. Self-descriptiveness is one important aspect of document-orientation. Ideally, a document will be modeled after real-world documents, such as purchase orders, invoices, or account statements. Documents should be designed so that they are useful on the context of a problem domain, which may suggest their use with one or more services.

Loosely coupled: Most SOA proponents will agree that loose coupling is an important concept. Unfortunately, there are many different opinions about the characteristics that make a system "loosely coupled". There are multiple dimensions in which a system can be loosely or tightly coupled, and depending on the requirements and context, it may be loosely coupled in some of them and tightly coupled in others. Dimensions include:

- **Time:** When participants are loosely coupled in time, they don't have to be up and running at the same time to communicate. This requires some way of buffering/queuing in between them, although the approach taken for this is irrelevant. When one participant sends a message to the other one, it does not rely on an immediate answer message to continue processing (neither logically, nor physically).
- **Location:** If participants query for the address of participants they intend to communicate with, the location can change without having to re-program, reconfigure or even restart the communication partners. This implies some sort of lookup process using a directory or address that stores service endpoint addresses.
- **Type:** In an analogy to the concept of static vs. dynamic and weak vs. strong typing in programming languages, a participant can either rely on all or only on parts of a document structure to perform its work.
- **Version**: Participants can depend on a specific version of a service interface, or be resilient to change (to a certain degree). The more exact the version match has to be, the less loosely coupled the participants (in this dimension).
- **Cardinality:** There may be a 1:1-relationship between service consumers and service providers, especially in cases where a request/response interaction takes place or an explicit message queue is used. In other cases, a service consumer (which in this case is more reasonably called a "message sender or "event source" may neither know nor care about the number of recipients of a message.
- **Lookup:** A participant that intends to invoke a service can either rely on a (physical or logical) name of a service provider to communicate with, or it can

perform a lookup operation first, using a description of a set of capabilities instead. This implies a registry and/or repository that is able to match the consumer's needs to providers capabilities (either directly or indirectly).

- **Interface:** Participants may require adherence to a service-specific interface or they may support a generic interface. If a generic interface is used, all participants consuming this generic interface can interact with all participants providing it. While this may seem awkward at first sight, the principle of a single generic (uniform) interface is at the core of the WWW's architecture.

Standards-compliant: A key principle to be followed in an SOA approach is the reliance on standards instead of proprietary APIs and formats. Standards exist for technical aspects such as data formats, metadata, transport and transfer protocols, as well as for business-level artifacts such as document types (e.g. in UBL). The most important aspect of any standard is its acceptance (which basically translates to "Microsoft needs to be on the author list" in case of Web services).

2.5 Service-Oriented Architecture Adoption

The adaptation of SOA continues to increase with more organizations implementing the principle of SOA into their business IT strategy. The service ontology acts as the connecting link between the service provider and the consumer of the service. Implementing SOA revolves around three key elements, the service provider, the registry and the consumer, the existence of all three can be within an organization. In a more general application, the elements might be from different organizations dealing with each other through the broker. Figure 2.2 shows the basic principle of the elements in SOA.

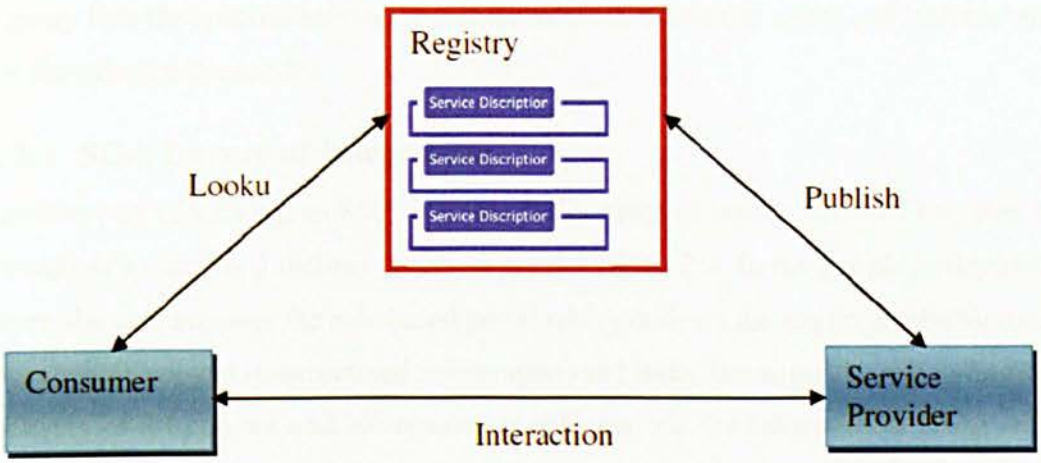

Figure 2.2: Simplified SOA Architecture [20]

Consumer: The application or the entity that uses the service, it can be an application or any form of software component that requires the need of a service. The consumer first requests the use of the service by locating the service, in the ontology of the service, over communication transport protocol such as SOAP (Simple Object Access Protocol) in a format that is acceptable to identify the specific service. Initiating and interaction is instantiated and executed when the service is located and negotiated with the provider.

Provider: The provider supplies the service that is available to the consumer; either a simple software function or a collection of services may constitute a given service. In order to use the service the provider publishes the service description information in the registry that serves as the criteria to be accessed by the consumer.

Registry: The registry is a collection of service descriptions that is available to be searched by consumers looking for a service to use. The registry is updated with the service information by the provider of the service. The registry does not initiate any form of connection between the consumer and the provider, only by making known what is available. The service consumer does not have access to the service from the provider directly and does not know the details of the service. The service provider publishes, using technology such as XML and web service, the service description to the ontology, either UDDI or other standard service. It is through this information that the consumer provides the specification for the service request to the registry. The

registry lists the specific services available to the consumer to select and interact with the provider for the service.

2.5.1 SOA Layers of Integration

Another way of viewing an SOA is to use the concept of layers. An SOA can then be thought of as having 5 distinct layers as per the figure 2.3. In the **people integration layer**, the user accesses the role-based portal which delivers the required collaboration, information, content (unstructured information) and tasks lists as per the role definition. Delivery of the content and information is achieved via the **information integration layer**. The task lists which comprise the outstanding tasks awaiting the user based on their role is delivered from the **process integration layer**. The actual interaction with the underlying **application layer**, both in terms of information delivery and process execution is handled by the **technical integration layer** [24].

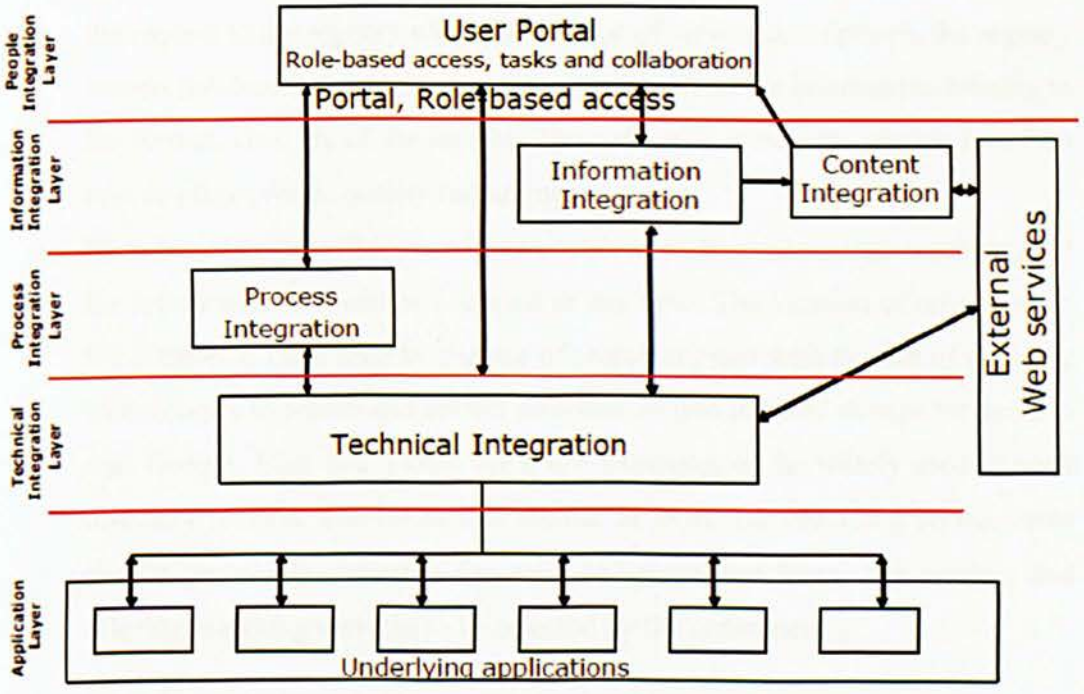

Figure 2.3: SOA — Layers of Integration [24]

Simple communication between the three, define the service location, selection and the use of the service by the consumer; more detailed information can be specified in the request to match directly to the service for the response to be accurate. Likewise, the provider of the service determines the information that is sent to the registry agent as

to the description to be advertised to the consumer. The whole process is based on the following steps:

- **Service Discovery**: In order to discover a service and execute it, the consumer must meet a standard that is set by the service provider, a contract or some form of agreed terms of the service. Communication between the consumer and registry must also conform to the SOA principle on communication technology, such as SOAP. Service discovery in SOA is simplified to three key players as explained above, however for the consumer to find, locate and interact with a provider to use a service, the service must be visible to the consumer and in so doing the consumer must be aware of the service, and be able to reach it [23].
- **Service Request:** The request is initiated by the consumer in a form of a transport protocol communication such as SOAP or HTTP with information as a standard SOA request format. In a simple request process, the consumer sends the request to the registry which has the list of service descriptions, the registry returns the descriptions based on the request with all the information relating to the format, cost, etc of the service. The consumer selects the service based on cost or other critical quality requirements.
- **Directory Service**: The internet has evolved to become a large resource pool for information that can be accessed at any time. The location of resources on the internet is facilitated by the use of search engines with the use of crawling technologies to search and collect information into indexed storage for users to use. Google, Bing and Yahoo are a few examples of the widely used internet directory service. The concept is similar in SOA, i.e. providing services that should be readily available for users to search and locate the service, and offering matching services to be selected by the consumer.

SOA's answer to directory services is UDDI (Universal Directory and Discovery Integration), JINI, [25] and others which have a different composition to the internet search engine; the provider of the service is expected to update the ontology with a description of the service containing the necessary details of requirement to use the service. Whereas internet search engines uses spider and other technologies to search for resources to update their index, the reverse is how UDDI works in SOA. JINI, on the other hand is based on the Java technology and requires Java virtual

machine (JVM) to be running on all instances that is used for service repository and discovery of services [26].

2.5.2 Challenges to Adoption

The main challenges to adoption of SOA include ensuring adequate staff training and maintaining the level of discipline required to ensure the services that are developed are reusable. Any technology, no matter how promising, can be abused and improperly used. Services have to be developed not simply for immediate benefit, but also (and perhaps primarily) for long-term benefit. To put it another way, the existence of an individual service isn't of much value unless it fits into a larger collection of services that can be consumed by multiple applications, and out of which multiple new applications can be composed. In addition, the definition of a reusable service is very difficult to get right the first time [24].

Another challenge is managing short-term costs. Building an SOA isn't cheap; reengineering existing systems costs money and the payback becomes larger over time. It requires business analysts to define the business processes, systems architects to turn processes into specifications, software engineers to develop the new code, and project managers to track it all [24]. Another challenge is that some applications may need to be modified in order to participate in the SOA. Some applications might not have a callable interface that can be service-enabled. Some applications are only accessible via file transfer or batch data input and output and may need additional programs for the purpose of service-enablement [24].

2.5.3 Characteristics of Successful SOA Implementations

Strong Executive Level Sponsorship and SOA Evangelist

Each project had strong sponsorship from high ranking individuals from the business and/or IT. This is critical for driving change throughout the organization and removing roadblocks. Without top-level support, many SOA initiatives never get the momentum, the resources and the drive required to allow IT to deliver the promise of SOA to the business. It was also noted that a strong SOA evangelist was highly

critical for each of these award-winning case studies. In fact, research shows that in instances where SOA evangelists leave a company, the company has a risk of failing with future projects or regressing back to the previous methods of delivering software [24].

Educating the Business of the Value of SOA

Each one of the case studies provided an enormous amount of value to the business. In some cases, the return on investment was several billions of dollars over the course of a few years. In order to find these extraordinary opportunities and to build a business case around them, it is critical that the business becomes educated on the promise of SOA. The key to educating the business, however, is not talking to the business about the technology or even mentioning the term service-oriented architecture. Instead the business needs to understand the key business drivers that are being addressed (quicker access to information, integration with customers and partners, eliminating wasteful business processes, etc.) on how IT has some "new methods" for helping to deliver these drivers. The business doesn't necessarily need to know how IT will do it; they need to understand which of their problems SOA solves and what is required from the business to help IT solve them [24].

Establish a Centre of Excellence (CoE)

Every winning case study had some form of CoE established. It may have been called something else, such as a Configuration Control Board, but all had some formal body that was responsible for governing the SOA initiative. Some of these companies already had in place an established Enterprise Architecture complete with IT governance and simply needed to make adjustments for SOA. Others did not have a formal governance plan and had to create one with enough controls in place to deliver the desired business results. The level of control and the scope of each company's governance model were unique, but every successful project sited governance as a key success factor [24].

Start With Well-defined Business Processes and Scale Up

In each case, candidate services were identified after well-defined business processes were established. In some cases, the business processes were already in place; in others some business processing reengineering was required prior to creating any

services. In each case, the goal was to start with some subset of business processes as opposed to trying to do it all at once. Each case study had a well-defined scope and a vision of what the future state looked like [24].

Define Completeness of Work within Services

A lot of thought was put into which services were critical to the key business drivers. Business services provided a complete business function. Most successful SOA implementations do not have a huge number of business services. This is where a lot of SOA projects run into trouble. They try to make everything into a service, whether it provides business value or not. There is a considerable amount of overhead and costs involved with building, governing, and maintaining services. Successful SOA implementations focus on a small number of core business services that provide real business value and don't waste precious time and money on services that don't have the payback [24].

Quality Assurance Is Key

SOA creates all sorts of new challenges for the QA department. Successful SOA implementations require that proper QA best practices, such as load testing of each service, is performed. Performance, security and governance testing should be a part of your overall testing plan to ensure that both the business and technical requirements are met [24].

ROI is Difficult to Achieve Initially and is Realized Over Time

SOA is not a technology; it is architecture. Like any other architectures, value is earned over time as the architecture expands and matures. Some of these companies were on their second or third SOA project and were realizing substantial ROI. Others were in their first iteration and did not see immediate ROI but instead were laying down the foundation for future SOA projects to maximize their ROI [24].

Deliver Substantial Business Value

In all cases, these award winning case studies delivered substantial business value. None of these case studies were focused on fixing IT infrastructure or based solely on reducing development costs through reuse. These may have been some side effects, but the value of the IT benefits are minuscule as compared to the business benefits

which in some cases were in the billions of dollars over a given time period. So for all of the pundits out there who claim that you should never talk to the business about SOA or that SOA is an IT initiative not a business initiative-look at the huge ROIs of these projects and the business transformations that occurred and reconsider those stances [24].

2.6 Service Oriented Architecture Governance

Governance has been rated as the main inhibitor of SOA adoption [26]. SOA governance provides a set of policies, rules, and enforcement mechanisms for developing, using, and evolving SOA-based systems, and for analysis of their business value. SOA governance includes policies, procedures, roles, and responsibilities for design time governance and runtime governance. Design-time governance includes elements such as rules for strategic identification of services, development, and deployment of services, reuse, and legacy system migration to services. It also enforces consistency in use of standards, SOA infrastructure, and processes.

SOA Governance is about ensuring that each new and existing service conforms to the standards, policies and objectives of an organization for the entire life of that service [27]. Also, SOA governance is the mechanism by which organizations ensure that their SOA implementation is built around the best possible alignment between the goals of the business and IT [28]. This definition of governance implies that one needs to have an SOA strategy, ensure that it's aligned with where the business is going, and develop a concrete idea of what to expect from SOA investments. To meet business, Enterprise Architecture (EA) and SOA goals, policies must be enacted across the different business areas: architecture, technology infrastructure, information, finance, portfolios, people, projects (or rather, the way in which projects are executed) and operations. This is the role of governance: i.e. policies, which need to be designed and enacted to ensure this alignment [27].

2.6.1 Why is SOA Governance needed?

SOA governance plays an increasingly important role in today's challenging business environment. It provides structure, commitment and support for the development,

implementation and management of SOA, as necessary, to ensure it achieves its objectives. SOA governance provides the following benefits:

- Realize business benefits of SOA
- Business process flexibility
- Improved time to market
- Maintaining Quality of Service (QoS)
- Ensuring consistency of service
- Measuring the right things
- Communicating clearly between businesses.

2.6.2 SOA Governance Implementation

SOA governance implementation is comprised of a number of sub-topics, which include IT alignment, SOA and IT governance [28 – 33]:

1. IT Alignment: The references in the area of alignment between business and technology provide support for discussion of all the strategies, technologies, results and outcomes that are presented in this study. Literature in the areas of SOA, SOA governance, IT governance and SOA implementation has roots in IT alignment.
2. IT Governance: Like SOA, IT governance is another foundational layer in the literature that needs to be covered before SOA governance can be appropriately discussed. IT governance aims to place a governance framework around new strategies and technologies such as SOA to ensure that they align with business goals. The convergence of SOA and IT governance results in the body of literature that is at the heart of the study SOA governance.
3. SOA: SOA is a core strategy and technology whose complexity spurred the need for SOA governance. The references related to SOA provide a foundational step before moving on to the discussion of SOA governance.
4. SOA Governance: The references in this section provide the basic framework for understanding SOA governance and the management goals that SOA governance aims to achieve.

5. SOA Governance Deployment: This section of references provides the majority of insight into the detailed deployment phases of a SOA governance program and is the most appealing to the audience of information technology managers who is the target for the final outcome of this study.
6. Methodology: This study outlines a method plan based on literature review and content analysis.

2.6.3 Role of Governance

The word *governance* describes many facets of management and policy for SOA-based systems. Depending on context, *governance* can refer to [24]:

- overseeing and enforcing policy (business design, technical design, application security) that directs the organization
- creating policy that directs the organization
- coordinating the people, policies, and processes that provide the framework for management decision-making
- taking action to optimize outcomes related to an individual's responsibility
- promoting efficiency in the organization
- determining the integrity of services.

Good governance exemplifies such characteristics as accountability, freedom of association and participation, a sound judicial system, freedom of information and expression, capacity building, and similar things. The concept of governance is not entirely new. What is new is that development and acquisition are, in an SOA context, very different, and management and policy apply to more than simply construction. However, like most aspects of management, traditional or otherwise, governance will be reflected in one or more processes.

2.6.4 Characteristics of Good SOA Governance

In the context of SOA, governance should encourage active and efficient use of available services by application builders. While a number of characteristics of

governance apply within the SOA context, we will focus in this column on just the following [24]:

- a flexible authority structure
- management incentives
- full operational life cycle.

2.7 Service-Oriented Architecture and Web Services

Web services are defined as "a family of technologies that consist of specifications, protocols, and industry-based standards that are used by heterogeneous applications to communicate, collaborate, and exchange information among them in a secure, reliable, and interoperable manner" [33]. Services in an SOA are modules of business or technical functionality with exposed interfaces to the functionality. Web services are the organizing principles of SOA at this time.

Although much has been written about SOA and Web services, there still is some confusion between these two terms among software developers. SOA is an architectural style, whereas Web services are a technology that can be used to implement SOAs. The Web services technology consists of several published standards, the most important ones being SOAP and WSDL. Other technologies may also be considered technologies for implementing SOA, such as CORBA. Although no current technologies entirely fulfil the vision and goals of SOA as defined by most authors, they are still referred to as SOA technologies. The relationship between SOA and SOA technologies is represented in Figure 2.4. Much of the technical information in this report is related to Web services technology, because it is commonly used in today's SOA implementations.

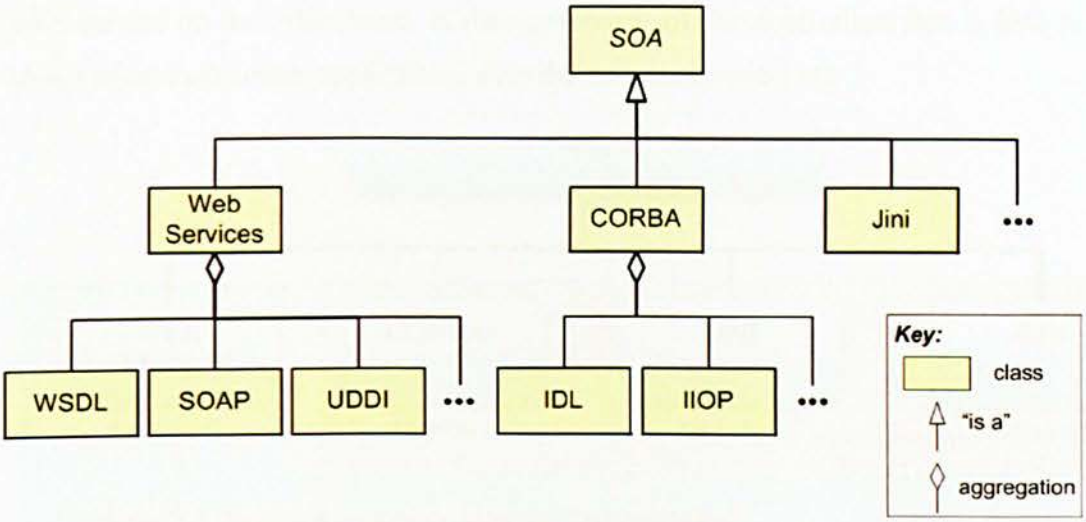

Figure 2.4: SOA and SOA Technologies [33]

2.7.1 Drivers for SOA

In large organizations, the following types of organizational, business, and technology changes drive a desire to reap the benefits of SOA:

- integration with legacy systems
- corporate mergers
- realignment of responsibilities through business reorganizations
- changing business partnerships (e.g., relationships with suppliers and customers)
- modernization of obsolete systems for financial, functional, or technical reasons
- acquisition or decommission of software products
- sociopolitical forces related to or independent of the drivers cited above.

2.7.2 XML Web Services

XML and Web services are used with each other in SOA, but the implementation of SOA does not necessarily mean the implementation of XML Web service. However, the implementation of XML Web service provides the foundation for the adaptation of distributed and integrated computing over the internet, forming part of the SOA architecture (figure 2.5). XML as mentioned above, offers the language format for platform independence using SOAP communication protocol over the internet, whereas

Web service on the other hand, is the component of the application that is able to communicate with other applications over the internet or web [34].

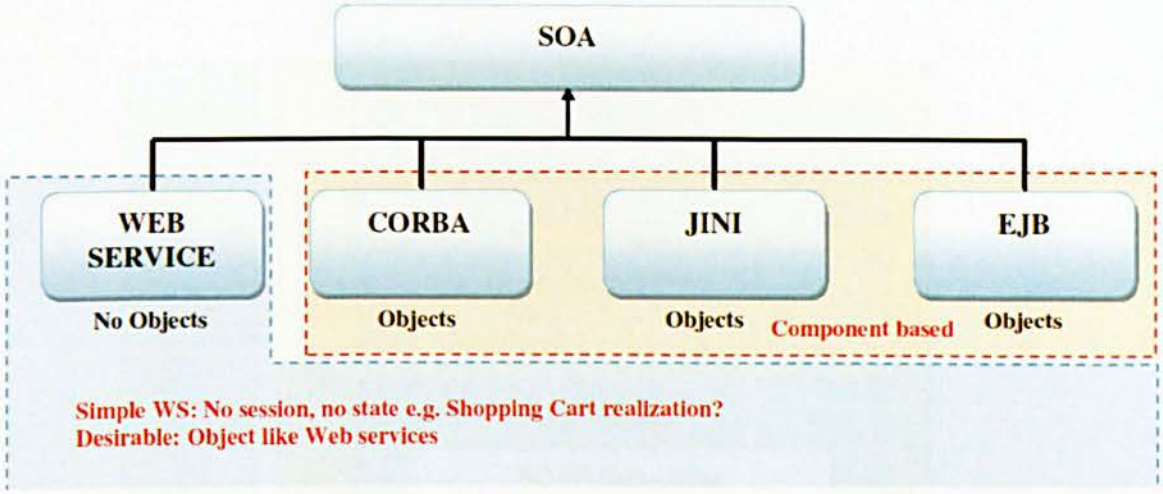

Figure 2.5: An Architecture of Software Which Is Composed of Services [34]

The means of communication and exchange of information between Web services is defined using XML. Fig 2.6 shows the simple Web service implementation, the communication between the component in the architecture are implemented in XML format [35].

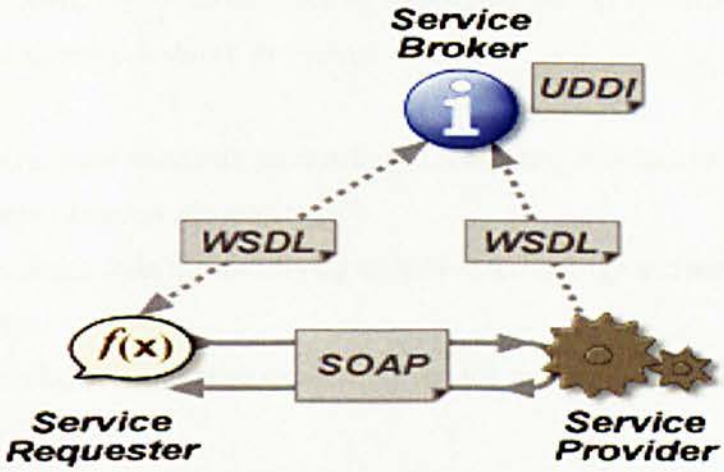

Figure 2.6: Web Service Architecture With XML [33]

Web service acts as the implementation tool for SOA; it has an interface that is described in the format of WSDL (Web Service Description Language). Figure 2.7 shows the details of a Web service, discovery of a service and service interface with a description of the service in a machine-processable format (WSDL). The Web service

interface is exposed for discovery allowing it to be used by other systems, using SOAP and communication through HTTP or SMTP protocols.

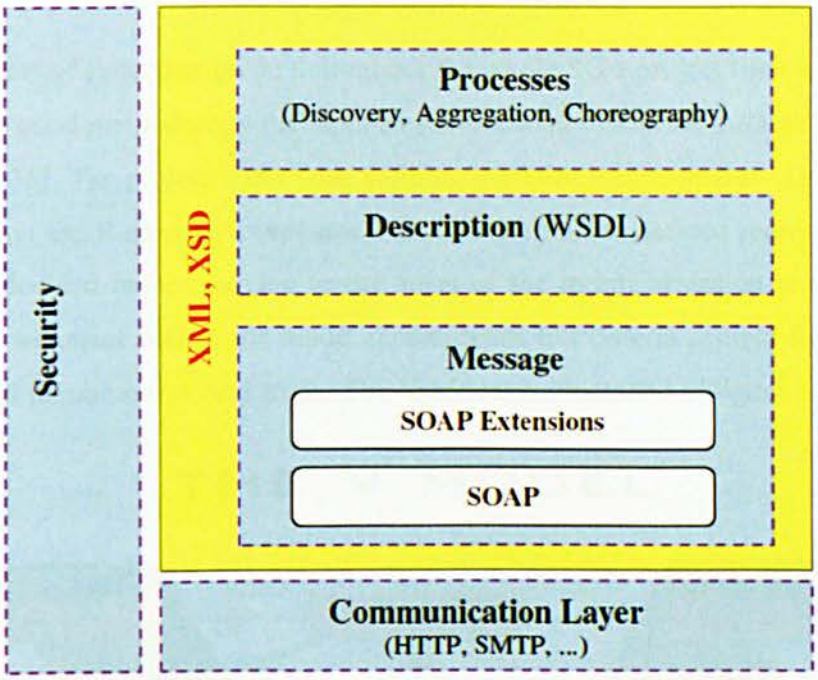

Figure 2.7: Web Services Architecture Stack [34]

Web services: XML-based technologies for messaging, service description, discovery, and extended features, providing:

- Pervasive, open standards for distributed computing interface descriptions and document exchange via messages.
- Independence from the underlying execution technology and application platforms.
- Extensibility for enterprise qualities of service such as security, reliability, and transactions.
- Support for composite applications such as business process flows, multi-channel access, and rapid integration.

2.8 Service-Oriented Architecture Strategies

SOA is the architectural approach that maintains loosely coupled services to allow business flexibility in an interoperable, technology-agnostic way. SOA involves a

composite set of business-aligned services that support a flexible and dynamically re-configurable end-to-end business processes realisation using interface-based service descriptions.

Some aspects of execution go on throughout the whole SOA project life cycle. The V-Model is a good methodology that applies some testing discipline through the project life cycle [36]. The project starts with defining the User Requirements. The V-Model suggests that the Business Acceptance Test Criteria for the defined requirements are clear and decided before moving to the onset of the technical design phase. Before moving to technical design, the model recommends test criteria distinct for that level of technical requirements, and so on. The V-Model is illustrated in figure 2.8.

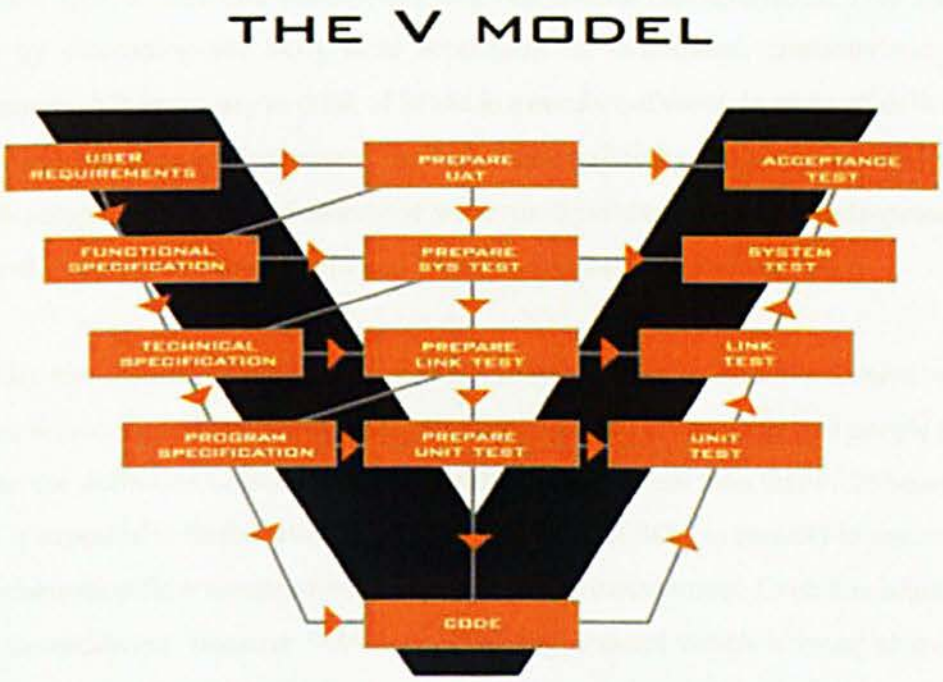

Figure 2.8: V-Model [36]

The work on SOA design defects has focused pattern and anti-pattern, some of which are discussed below [36]:

1. **Percolating Process:** Organisations start with a detailed Process map and then attempt to "fit" this into an SOA; this refactoring leads to the process becoming

the dominant feature and leads to a process-oriented architecture (POA) rather than SOA.

2. **Point to Point Web Services:** Web service point to point is *still* point to point; doing a bad practice in XML does not make it better.
3. **Splitting Hairs:** Splitting into two separate tiers of Service and Process, with separate rules and governance results in divergent solutions.
4. **IT2B:** Creating "business" services based on the belief that IT understands the business results in services that neither meet IT nor business goals.

2.9 Summary

SOA is a type of software architecture that has special characteristics. This chapter began by discussing the SOA field describing its definitions, characteristics and motivations. It is necessary to think of SOAs in a number of ways. In terms of delivering design and run-time environments, the key characteristics of process orientation, an agile development toolkit and enterprise wide run-time management is underpinned by effective programme management and an agile development methodology.

An SOA can also be thought of in a more abstract way; as an environment which delivers service-based integration at application, process, information and people level. Further, the definition of SOA today is significantly different than that of 25 years ago which is expected to further evolve. It is a logical step for SOA eventually to encompass both information flow management and organisation management. Even this is unlikely to be the end-point. Because SOA is an evolving concept which is being shaped by emerging standards and technologies the definition of what SOA encompasses will also, inevitably, continue to evolve and to expand.

The Web services technology, which is the most appropriate environment to develop SOA currently, was also mentioned. However, it was emphasized that the Web services technology is not the only possible way to develop SOA concepts, which are technology independent. So, we can see that the Web services standards, both the core and extended specifications, contribute significantly to the ability to create and maintain SOAs on which to build new enterprise applications. These applications are often called composite applications because they work through a combination of multiple services.

We've seen that SOA is not an end in itself but a preparation for a longer journey. It's a set of maps and directions to follow that lead to a better IT environment. It's a blueprint for an infrastructure that aligns IT with business, saves money through reuse of assets, rapid application development, and multichannel access.

Web services have had an initial success with the core standards and are now on to the next step in the journey, which is to define extended features and functions that will support more and more kinds of applications. Service orientation provides a different perspective and way of thinking than object orientation. It's as significant a change as going from procedure-oriented computing to object-oriented computing. Services tend toward complementary rather than replacement technology, and are best suited for interoperability across arbitrary execution environments, including object oriented, procedure oriented, and other systems.

In summary, SOA promotes loose coupling, function specific solution and platform independence. A few of SOA characters which also fall in the domain of distributed computing applications development. Although SOA has been used in industry for a while, interest in SOA has resurfaced strongly with the deployment of SOA in Cloud Computing. But SOA is not the solution to all problems linked with software development. There are a lot of problems: Ranging from finding the required services, providing acceptable performance, security, realising transactions up to maintaining one's own service, even if foreign, integrated services have changed or are closed. There are a lot of problems to resolve, but there are a lot of possibilities too. It will depend on industry and academia, to develop an overall answer, containing solutions to all of these problems.

In summary, SOA is a software architecture that determines the features that make up an application and should be made available as services that communicate with each other through messages. That way, applications can be developed into small parties, facilitating management of development teams. But SOA is more than that — it is a robust architecture, focused on the integration of systems, whose main benefits are discussed below:

Table 2.1: SOA Features and Benefits

Feature	Benefits
Service	• Improved information flow • Ability to expose internal functionality • Organizational flexibility
Service Reuse	• Lower software development and management costs
Messaging	• Configuration flexibility
Message Monitoring	• Business intelligence • Performance measurement • Security attack detection
Message Control	• Application of management policy • Application of security policy
Message Transformation	• Data translation
Message Security	• Data confidentiality and integrity
Complex Event Processing	• Simplification of software structure • Ability to adapt quickly to different external environments • Improved manageability and security
Service Composition	• Ability to develop new function combinations rapidly
Service Discovery	• Ability to optimize performance, functionality, and cost • Easier introduction of system upgrades
Asset Wrapping	• Ability to integrate existing assets
Virtualization	• Improved reliability • Ability to scale operations to meet different demand levels
Model-driven Implementation	• Ability to develop new functions rapidly

CHAPTER 3: SOFTWARE QUALITY

3.1 Introduction

Software quality measurement is about quantifying to what extent software design possesses desirable characteristics. To understand the landscape of software quality it is central to answer the so often asked question: what is quality? The following are the quality principles according to quality management gurus:

Quality according to Crosby: The word "quality" is often used to signify the relative worth of something in such phrases as "good quality", "bad quality" and "quality of life" which means different things to each and every person. As follows quality must be defined as conformance to requirements' if we are to manage it. Consequently, the nonconformance detected is the absence of quality, quality problems become nonconformance problems, and quality becomes definable [37].

Quality according to Deming: The problem inherent in attempts to define the quality of a product, almost any product, where stated by the master Walter A. Shewhart. The difficulty in defining quality is to translate future needs of the user into measurable characteristics, so that a product can be designed and turned out to give satisfaction at a price that the user will pay. This is not easy, and as soon as one feels fairly successful in the endeavor, he finds that the needs of the consumer have changed, competitors have moved in etc [38].

Quality according to Feigenbaum: Quality is a customer determination, not an engineer's determination, not a marketing determination, nor a general management determination. It is based upon the customer's actual experience with the product or service, measured against his or her requirements stated or unstated, conscious or merely sensed, technically operational or entirely subjective and always representing a moving target in a competitive market [39].

Quality according to Ishikawa: We engage in quality control in order to manufacture products with the quality which can satisfy the requirements of consumers. The mere

fact of meeting national standards or specifications is not the answer, it is simply insufficient. International standards established by the International Organization for Standardization (ISO) or the International Electrotechnical Commission (IEC) is not perfect. They contain many shortcomings. Consumers may not be satisfied with a product which meets these standards. We must also keep in mind that consumer requirements change from year to year and even frequently updated standards cannot keep the pace with consumer requirements. How one interprets the term "quality" is important. Narrowly interpreted, quality means quality of products. Broadly interpreted, quality means quality of product, service, information, processes, people, systems etc. [40].

Quality according to Juran: The word quality has multiple meanings. Two of those meanings dominate the use of the word: 1) Quality consists of those product features which meet the need of customers and thereby provide product satisfaction. 2) Quality consists of freedom from deficiencies. Nevertheless, in a handbook such as this it is most convenient to standardize on a short definition of the word quality as "fitness for use" [41].

Quality according to Shewhart: There are two common aspects of quality: One of them has to do with the consideration of the quality of a thing as an objective reality independent of the existence of man. The other has to do with what we think, feel or sense as a result of the objective reality. In other words, there is a subjective side of quality [42].

Software maintenance, which means fixing any defects after delivery, can cost 90% of the total cost of normal software projects [43]. Many studies have discussed that any new function to improve the software quality or fixing defects are the major parts of those costs [44]. According to previous studies focused on bad design practice, which also labeled defects as anti-patterns, smells or anomalies, these bad practices are sometimes unavoidable and should be prevented by the development teams as early as possible [45].

Software defects compromise the operation of the resultant software system. It is thus a good practice to deal with the bugs at an early stage so as to avoid the resulting

weaknesses. Working with defects inherited from the preliminary periods of design time causes a lot of problems as the debugging becomes cumbersome, more so in finding errors that originates from the requirements and design [46]. Missing requirements, incomplete requirements, code logic error, wrong requirements, conflicting code modules, conflicting requirements and requirement execution cause great problems, with the first being the worst case. This is the reason why defects at earlier stages of development must be removed at as early a stage as possible.

This chapter starts with quality in general; SOA quality and quality attributes are discussed in section 3.2. Section 3.3 presents a detailed discussion of quality models, whereas section 3.4 presents a detailed discussion of quality metrics of SOA. Section 3.5 presents a summary of the chapter.

3.2 Quality of Service-Oriented Architecture

SOA-based software development has been gaining momentum in recent years due to its perceived advantages such as more flexibility, and heterogeneity in software structure and design. Moreover, SOA facilitates reusability through the encapsulation of software products inside *services*. With the growing trend of deploying SOA in the IT infrastructure of large companies, it is imperative that the performance and quality of the products delivered in such a context is clearly evaluated and guaranteed accordingly. Quality is currently considered one of the main assets with which a firm can enhance its competitive global position. This is one reason "why quality has become essential for ensuring that a company's products and processes meet customers' needs" [47].

3.2.1 Hierarchy of Quality

The sum of quality characteristics and attributes applied to a Web service is defined as "the totality of features and characteristics of a product or service that bear on its ability to satisfy stated or implied needs" [48]. Kan [49] defined the Quality concepts as Pyramids, its base are quality metrics and its top are quality characteristics as shown in figure 3.1.

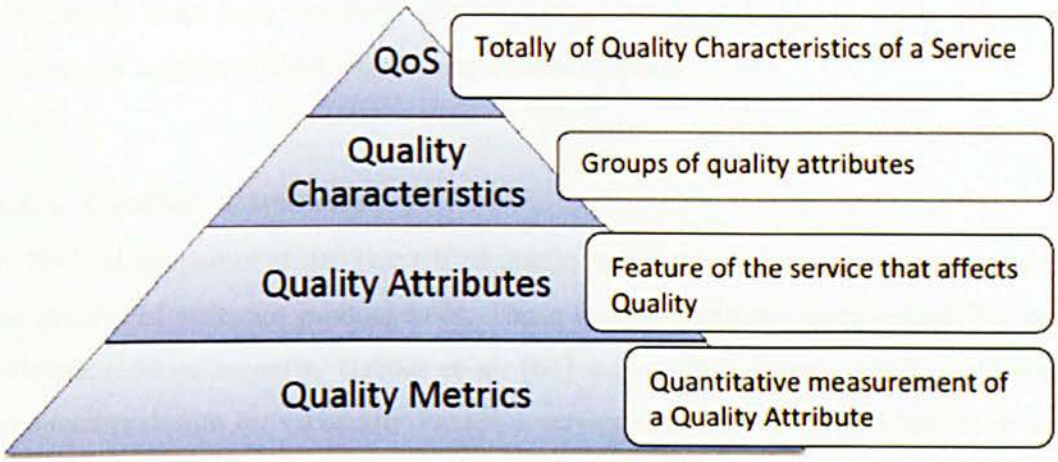

Figure 3.1: Hierarchy of Quality Concepts [49]

One of the most challenging aspects of building SOA applications is quality assurance. Developers must analyze every flow path, every condition, and every fault to ensure that processes are bullet-proof. A software quality attribute of a software system is a characteristic, feature or property that describes part of the software system. Internal software quality attributes reflect structural properties of a software system (e.g.: software size in terms of Lines of Code) [50].

3.2.2 Quality Assurance in Service-Oriented Architectures

Quality assurance has a vital role in building a software system because it provides confidence and lowers the risks associated. Assurance in service-oriented systems is a challenging problem, which requires a flexible and dynamic solution. When we talk about quality of a service-oriented system we have to consider all the included services that are interdependent to provide that service, including all the limitations of resources and runtime situation.

It is anticipated that due to the difference between the nature of traditional software development technologies and SOA that the verification and validation process in the quality assurance model will resemble a paradigm shift. For instance, while a group of independent software quality assurance experts can validate a traditional software product based on structural (white-box) or functional (black-box) testing techniques; in contrast such a process needs to be carried out through the collaboration of service providers, brokers, and clients in an SOA setting. Moreover, while service providers

can benefit from both structural and functional testing techniques, the brokers and clients will only have black-box testing at their disposal.

3.2.3 Quality Attributes

In 2011 Montagud et al. [61] classified quality attributes and measures for assessing the quality of software product lines. These quality attributes were reusability and efficiency. More recently, Galster et al. [62] suggested a framework for reference architecture design for variability-intensive service-based systems using the following quality attributes: Variability, Scalability, Interoperability, Performance, Reliability, Privacy and Security. Marko [63] suggested an SOA prototype to evaluate the quality of the architecture resulting from the combination of EBI and SOA patterns. The quality is evaluated with respect to: Efficiency, Functionality, Maintainability, Portability, Reliability and Usability. Table 3.1 shows a summary of SOA Quality Attributes and their definitions.

Table 3.1: SOA Quality Attributes

SOA Quality Attributes	Definition
Adaptability	The quality of being adaptable to changes. The use of an SOA approach should have a positive impact on adaptability, as long as the adaptations are managed properly. However, the management of this quality attribute is left up to the service users and providers, and no standards exist to support it. This attribute must be managed in coordination with other quality attributes including stability, performance, and availability, and the necessary trade-offs must be identified and made. Adaptability means the ease with which a system may be changed to fit changed requirements. Adaptability for a business means it can adapt quickly to new opportunities and potential competitive threats, which implies that the application development and maintenance groups within the business can quickly change the existing systems.
Analysability	The capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified.
Auditability	Auditability is the quality factor representing the degree to which an application or component keeps sufficiently adequate records to support one or more specified financial or legal audits. With the ever-increasing need for systems to comply with business and regulatory legislation (financial and health sectors especially), the ability to audit a system for compliance is an important consideration. However, the flexibility offered by SOAs may make such audits difficult. If an application using an SOA approach dynamically uses external services, it may be difficult to track which services are actually used. If an outside service uses additional services (i.e., is composed of other services) to carry out its functionality, the audit process becomes even more complex.

Availability	Availability refers to the ability of the user community to access the service, whether to submit a new request, update or alter an existing request, or collect the results of a previous request. If a user cannot access the service, it is said to be *unavailable*.
Changeability	The capability of the software product to enable a specified modification to be implemented.
Correctness	Accountability for satisfying all requirements of the system. Measure of exact adherence to specifications.
Efficiency	The *efficiency* characteristic relates to the capability of a test specification to provide acceptable performance in terms of speed and resource usage. The ISO/IEC 9126 subcharacteristics *time behaviour* and *resource utilisation* apply without change.
Extensibility	Extensibility is the ease with which the services' capabilities can be extended without affecting other services or parts of the system. Extensibility for architecture today (in particular, an SOA) is important because the business environment in which a software system lives is continually changing and evolving. These changes in the environment will mean changes in the software system, service users, and service providers and the messages exchanged among them. Extending an SOA by adding new services or incorporating additional capabilities into existing services is supported within an SOA. However, the interface/formal contract must be designed carefully to make sure that it can be extended, if necessary, without causing a major impact on the service users.

Interoperability	The ability to exchange and use information (usually in a large heterogeneous network made up of several local area networks). Interoperability may occur between two (or more) entities that are related to one another in one of three ways: **Integrated:** where there is a standard format for all constituent systems **Unified:** where there is a common meta-level structure across constituent models, providing a means for establishing semantic equivalence **Federated:** where models must be dynamically accommodated rather than having a predetermined meta-model.
Maintainability	*Maintainability* of test specifications is important when test developers are faced with changing or expanding a test specification. It characterizes the capability of a test specification to be modified for error correction, improvement, or adaption to changes in the environment or requirements. The *analysability*, *changeability*, and *stability* sub-characteristics from ISO/IEC 9126 are applicable to test specifications as well. The *testability* sub-characteristics do not play any role for test specifications.
Modifiability	Modifiability is the ability to make changes to a system quickly and cost-effectively. Modifiability considers how the system can accommodate anticipated and unanticipated changes and is largely a measure of how changes can be made locally, with little ripple effect on the system at large. The world around most software systems is constantly changing. This requires software systems to be modified several times after their initial development.

Performance	Performance refers to the system responsiveness: either the time required responding to specific events, or the number of events processed in a given time interval. An SOA approach can have a negative impact on the performance of an application due to network delays, the overhead of looking up services in a directory, and the overhead caused by intermediaries that handle communication. The service user must design and evaluate the architecture carefully, and the service provider must design and evaluate its services carefully to make sure that the necessary performance requirements are met.
Reliability	The reliability characteristic describes the capability of a test specification to maintain a specific level of performance under different conditions. In this context, the word "performance" expresses the degree to which needs are satisfied. The reliability sub-characteristics maturity, fault-tolerance, and recoverability of ISO/IEC 9126 apply to test specifications as well. However, new sub-characteristics test repeatability and security has been added. Test results should always be reproducible in subsequent test runs if generally possible. Otherwise, debugging the SUT to locate a defect becomes hard to impossible. Test repeatability includes the demand for deterministic test specifications.
Reusability	Although *reusability* is not part of ISO/IEC 9126, we consider this aspect to be particularly important for test specifications since it matters when test suites for different test types are specified. For example, the test behaviour of a performance or stress test specification may differ from a functional test, but the test data, such as predefined messages, can be reused between those test suites. It is noteworthy that the sub-characteristics correlate with the *maintainability* aspects to some degree. *Reusability* is the degree to which a software module or other work product can be used in more than one computing program or software system.

Scalability	Scalability is the ability of SOA to function well when the system is changed in size or in volume in order to meet users' needs. Extending an SOA by adding new services or incorporating additional capabilities into existing services is supported within an SOA. However, the interface/formal contract must be designed carefully to make sure that it can be extended, if necessary, without causing a major impact on the service users.
Security	The need for encryption, authentication, and trust within an SOA approach requires detailed attention within the architecture. Many standards are being developed to support security, but most are still immature. If these issues are not dealt with appropriately within the SOA, security could be negatively impacted. Due to the distributed nature of the current enterprise systems, we have difficulty in administering security policies and bridging diverse security models. This leads to increased opportunities to make mistakes and leave security holes; hence the chance of accidental disclosure and the vulnerability to attack goes up.
Stability	The capability of the software product to avoid unexpected effects from modifications of the software.
Testability	Testability is the degree to which a system or service facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met. Testability can be negatively impacted when using an SOA due to the complexity of the testing services that are distributed across a network. Those services might be provided by external organizations where access to the source code is not available, and if they implement runtime discovery of services, it may be impossible to identify which services are used until a system executes. It is up to the service users and providers to test the services, and very little support is currently provided for the end-to-end testing of an SOA.

Understandability	The degree to which the purpose of the system or component is clear to the evaluator. *Understandability* is important since the test user must be able to understand whether a test specification is suitable for his needs. Documentation and description of the overall purpose of the test specification are key factors – also to find suitable test selections.
Usability	The *usability* attributes characterise the ease to actually instantiate or execute a test specification. This explicitly does not include usability in terms of difficulty to maintain or reuse parts of the test specification which are covered by other characteristics. Usability may decrease if the services within the application support human interactions with the system and there are performance problems with the services. It is up to the services users and providers to build support for usability into their systems.
Data Granularity	In SOAs, service users and service providers communicate over a network—a process that can introduce delays, possibly in the order of seconds, in user interactions. To avoid these delays, not only must the service respond to user requests with the data requested but also with other relevant data that may not be immediately displayed.
Operability and Deployability	Operating and deploying services and systems that use services need to be managed carefully to avoid problems. The interactions and tradeoffs among this and other quality attributes need to be monitored and managed.

3.3 Quality Metrics of Service-Oriented Architecture

A software metric is an algorithm which computes a numeric value from source code to measure properties of a software system. The purpose of software metrics is to make assessments throughout the software life cycle as to whether the software quality requirements are being met [62]. Software metrics are often used to assess the ability of software to achieve a predefined goal [63]. Software metrics are a means of steering development processes by assessing the quality of a software product and finding imbalances in the code base. Metrics have always been used to help guide managers with decisions about their organizations. Metrics have always been used to help guide

managers with decisions about their organizations. Defect tracking, for example, has traditionally been a metric used to measure software quality throughout the lifecycle.

3.4 Quality Models of Service-Oriented Architecture

Software quality attributes are benchmarks that describe the intended behaviour of a system within its environment. Quality attributes (QA) is defined as "A feature or characteristic that affects an item's quality"[51].

One of the most important quality models is the quality model presented by McCall et al. [52]. They presented a quality model focusing on a number of software quality factor that reflect both the users' views and the developers priorities. The main quality factors were correctness, reliability, efficiency and integrity as shown in figure 3.2.

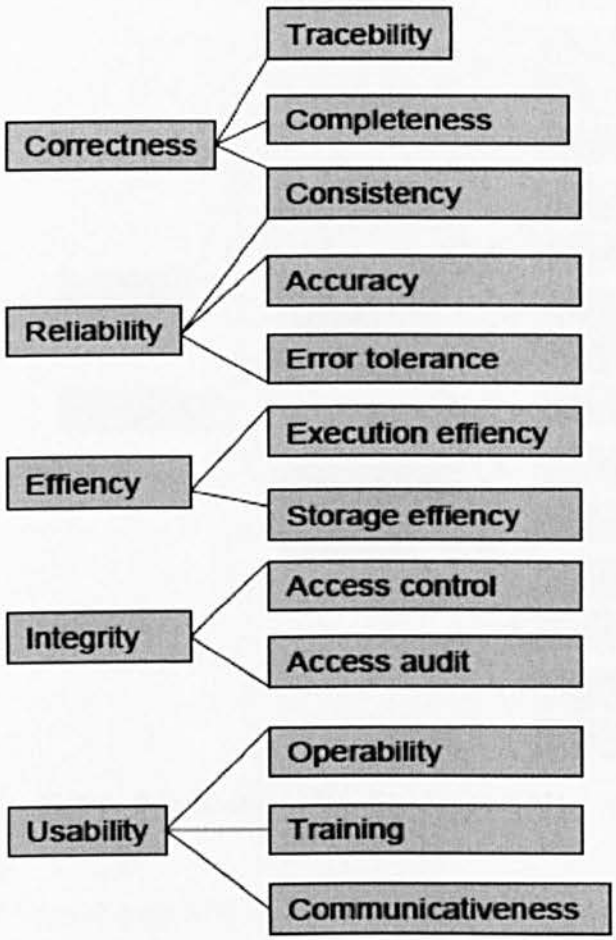

Figure 3.2: McCall Quality Model [52]

The second basic quality model is the quality model presented by Boehm et al. [53]. Boehm's model is similar to the McCall quality model in that it also presents a hierarchical quality model consisted of 7 quality factors Portability, Reliability, Efficiency, Usability, Testability, Understandability and Flexibility as shown in figure 3.3.

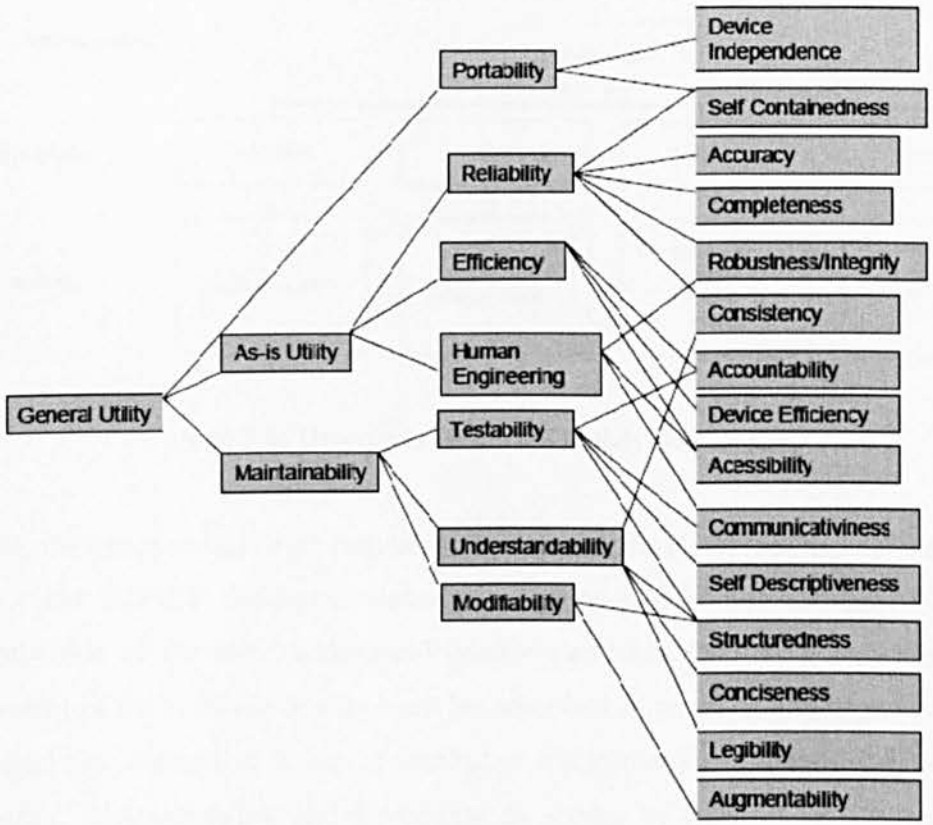

Figure 3.3: Boehm's Quality Model [53]

In 1987, Grady and Caswell proposed the FURPS model. It takes into account the five characteristics: Functionality, Usability, Reliability, Performance, and Supportability. When the FURPS model is used, two steps are considered: setting priorities and defining quality attributes that can be measured. Grady and Caswell noted that setting priorities is important given the implicit trade-off, i.e. one quality characteristic can be obtained at the expense of another. One disadvantage of this model is that it fails to take account of the software product's portability [54].

In 1996, Dromey [55] proposed a product based quality model that recognizes that quality evaluation differs for each product and that a more dynamic idea for modelling the process is needed to be wide enough to apply for different systems. Dromey was focusing on the relationship between the quality attributes and the sub-attributes, as well as attempting to connect software product properties with software quality attributes. Figure 3.4 shows Dromey's generic quality model.

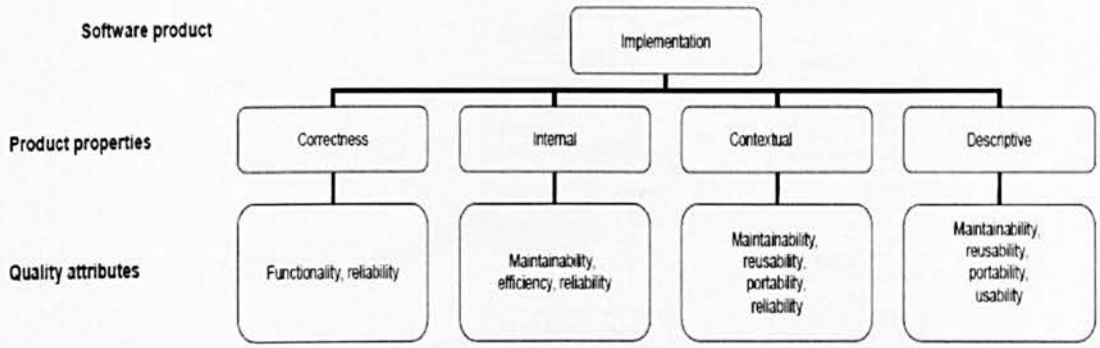

Figure 3.4: Dromey's Generic Quality Model [55]

In 1998, the International Organization for Standardization (ISO) [56] had defined a set of ISO and ISO/IEC standards related to software quality. The ISO/IEC 9126 is currently one of the most widespread quality standards. ISO 9126 indicated that component of the software quality must be described in terms of one or more of six characteristics defined as a set of attributes: Functionality, Reliability, Usability, Efficiency, Maintainability and Portability as shown in figure 3.5. Wiegers [57] suggested fourteen quality attributes: Reliability, Usability, Integrity, Efficiency, Portability, Reusability, Interoperability, Maintainability, Flexibility, Testability, Robustness, Installability, Safety and Availability.

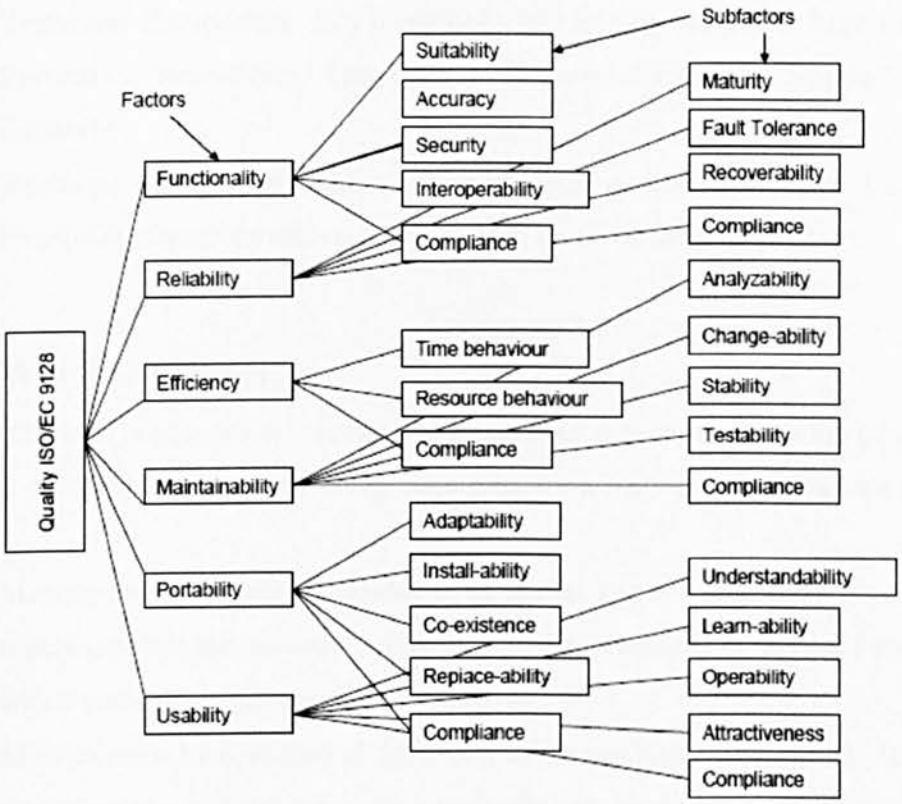

Figure 3.5: ISO Quality Model [57]

Also in 2003, Ortega et al. [47] designed a model prototype that reflects the essential attributes of quality. This model pinpointed three working areas based on McCall's Quality model as follows:

- **Product Operation**: which referred to the product's ability to be quickly understood, efficiently operated and capable of providing the results required by the user; the following quality attributes were taken into consideration: Modifiability, Reliability, Efficiency, Integrity, and Usability.
- **Product Revision**: which related to error correction and system adaptation, the following quality attributes were taken into consideration: Maintainability, Flexibility and Testability.
- **Product Transition**: which contained the following quality attributes: Portability, Reusability and Interoperability.

Recently, Pettersson [58] created an SOA Quality Evaluation Model that was applicable to SOA implementations. The model was based on two perspectives:

- **Technical Perspective**: this contained eight quality attributes: Modifiability, Portability, Reusability, Integrability, Security, Efficiency, Scalability and Reliability.
- **Business Perspective**: this contained four quality attributes: Usability, Flexibility, Development costs and Return on Investment (ROI).

3.4.1 Why Use Metrics?

Metrics allow an organization to identify the causes of defects that have the greatest effect on software development. The ground rules for a Metrics Usage Plan are that:

- **Metrics must be understandable to be useful**. For example, lines-of-code and function points are the most common, accepted measures of software size with which software engineers are most familiar.
- **Metrics must be economical**. Metrics must be available as a natural by-product of the work itself and integral to the software development process. Studies indicate that approximately 5% to 10% of total software development costs can be spent on metrics. The larger the software program, the more valuable the investment in metrics becomes. Therefore, do not waste programmer time by requiring specialty data collection that interferes with the coding task. Look for tools which can collect most data on a non-intrusive basis.
- **Metrics must be field tested**. Beware of software contractors who offer metrics programs that appear to have a sound theoretical basis, but have not had practical application or evaluation. Make sure proposed metrics have been successfully used on other programs or are prototyped before accepting them.
- **Metrics must be highly leveraged**. You are looking for data about the software development process that permit management to make significant improvements. Metrics that show deviations of .005% should be relegated to the trivia bin.
- **Metrics must be timely**. Metrics must be available in time to effect change in the development process. If a measurement is not available until the program is in deep trouble it has no value.
- **Metrics must give proper incentives for process improvement**. High scoring teams are driven to improve performance when trends of increasing

improvement and past successes are quantified. Conversely, metrics data should be used very carefully during contractor performance reviews. A poor performance review, based on metrics data, can lead to negative government/industry working relationships. *Do not use metrics to judge team or individual performances.*

- **Metrics must be evenly spaced throughout all phases of development.** Effective measurement adds value to all life cycle activities.
- **Metrics must be useful at multiple levels.** They must be meaningful to both management and technical team members for process improvement in all facets of development.

3.4.2 Quality Metrics

Burnstein [64] defined quality metrics as "a quantitative measurement of the degree to which an item possesses a given quality attribute". Different metrics have been proposed for object-oriented systems. Several prior studies [50, 66 & 86] had used metrics to identify defects impact on quality attributes, these metrics are:

3.4.2.1 Size Metrics:

Size of a class is used to evaluate the ease of understanding of code by developers and maintainers. Size can be measured in a variety of ways. These include counting all physical lines of code, the number of statements, the number of blank lines, and the number of comment lines. Size metrics is often measured using the Lines-Of-Code metric (LOC). Cardoso et al. [87] proposed the number of basic activities (NOA) as an alternative to count the LOC metric.

Lines-of-Code metric (LOC): which counts the number of statements within a program source code

Impact on quality: Size metrics are good candidates for developing cost or effort estimates for implementation, review, testing, or maintenance activities. Such estimates are then used as input for project planning purposes and the allocation of personnel.

3.4.2.2 Coupling Metrics:

Coupling metrics measure the relationships between entities [68 - 90]:

Depth of Inheritance Tree (DIT): The depth of a class within the inheritance hierarchy is the maximum number of steps from the class node to the root of the tree and is measured by the number of ancestor classes as shown in figure 3.6. The deeper a class is within the hierarchy, the greater the number of methods it is likely to inherit making it more complex to predict its behavior. Deeper trees constitute greater design complexity, since more methods and classes are involved, but the greater the potential for reuse of inherited methods. A support metric for DIT is the number of methods inherited (NMI).

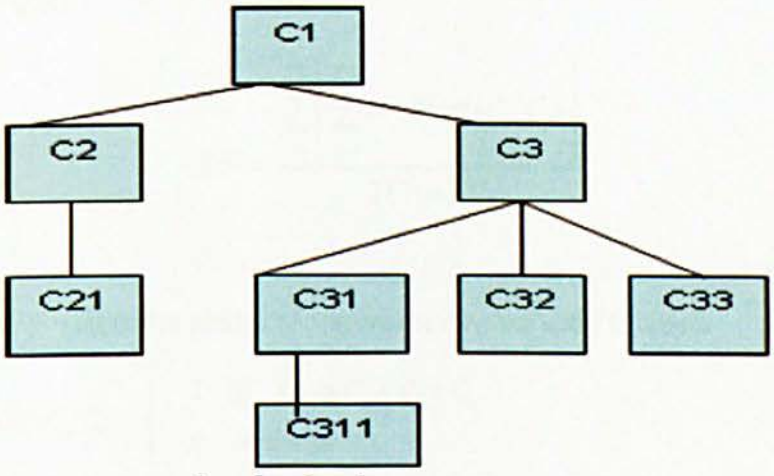

Figure 3.6: Depth of Inheritance [79]

Response Set For a Class (RFC): The RFC is the count of the set of all methods that can be invoked in response to a message to an object of the class or by some method in the class. This includes all methods accessible within the class hierarchy. This metric looks at the combination of the complexity of a class through the number of methods and the amount of communication with other classes. The larger the number of methods that can be invoked from a class through messages, the greater the complexity of the class. If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes complicated since it requires a greater level of understanding on the part of the tester. A worst case value for possible responses will assist in the appropriate allocation of testing time.

Coupling between Object Classes (CBO) / Coupling Object Factor (COF): is a count of the number of other classes to which a class is coupled. It is measured by counting the number of distinct non-inheritance related class hierarchies on which a class depends. It used to compare the level of coupling between classes. Excessive coupling is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is to reuse in another application. The larger the number of couples, the higher the sensitivity to changes in other parts of the design and therefore maintenance is more difficult. Strong coupling complicates a system since a class is harder to understand, change or correct by itself if it is interrelated with other classes. Complexity can be reduced by designing systems with the weakest possible coupling between classes. This improves modularity and promotes encapsulation. The following formula can be used:

$$COF = \frac{\sum_{i=1}^{TC}\left[\sum_{j=1}^{TC} is\_client(C_i, C_j)\right]}{TC^2 - TC} \qquad \textbf{Eq. (3.1)}$$

where:

$TC^2 - TC$ = maximum number of couplings in a system with TC classes

$$is\_client(C_c, C_s) = \begin{cases} 1 & iff \quad C_c \Rightarrow C_s \wedge C_c \neq C_s \\ 0 & otherwise \end{cases}$$

Impact on quality: Coupling connections cause dependencies between design elements, which, in turn, have an impact on system qualities such as maintainability (a modification of a design element may require modifications to its connected elements) or testability (a fault in one design element may cause a failure in a completely different, connected element). Thus, a common design principle is to minimize coupling.

3.4.2.3 Complexity Metrics:

Complexity is a measure of the degree of difficulty in understanding and comprehending the internal and external structure of classes and their relationships. Complexity metrics measure complexity in terms of control constructs and lexical tokens, respectively [68, 80, 81 & 87].

Source Line of Code (SLOC): the number of executable lines of source code.

Weighted Methods per Class (WMC): The WMC is a count of the methods implemented within a class or the sum of the complexities of the methods (method complexity is measured by cyclomatic complexity). The second measurement is difficult to implement since not all methods are assessable within the class hierarchy due to inheritance. The number of methods and the complexity of the methods involved is a predictor of how much time and effort is required to develop and maintain the class. The larger the number of methods in a class, the greater the potential impact on children; children inherit all of the methods defined in the parent class. Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse. The following formula can be used:

$$WMC\ (c) = \sum_{m \in M_{Im}(c)} VG\ (m) \qquad \text{Eq. (3.2)}$$

VG is the McCabe's Cyclomatic Complexity (CC)

$VG = 2 + ifs + loops + switch\ cases - return$

Good predictor of how much time/effort is required to: implement the class and maintain the class.

Number of Children (NOC): The number of children is the number of immediate subclasses subordinate to a class in the hierarchy as shown in figure 3.7. It is an indicator of the potential influence a class can have on the design and on the system. The greater the number of children, the greater the likelihood of improper abstraction of the parent and may be a case of misuse of sub-classing. But the greater the number of children, the greater the reuse since inheritance is a form of reuse. If a class has a large number of children, it may require more testing of the methods of that class, thus increase the testing time.

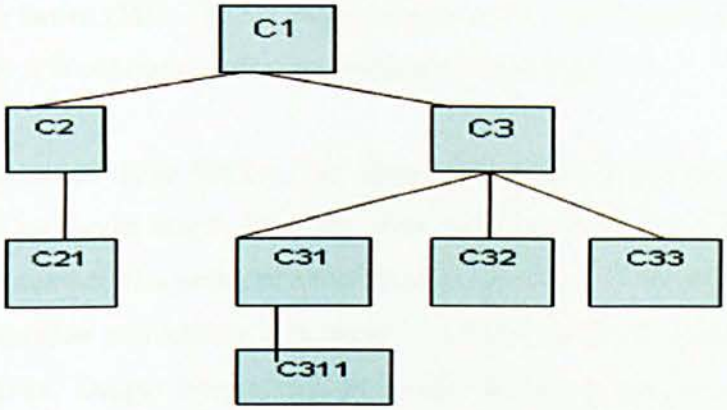

Figure 3.7: Number of Children [81]

Cyclomatic Complexity (CC): Used to evaluate the complexity of an algorithm in a method. It is a count of the number of test cases that are needed to test the method comprehensively. A method with a low cyclomatic complexity is generally better. This may imply decreased testing and increased understandability or that decisions are deferred through message passing, not that the method is not complex. Cyclomatic complexity cannot be used to measure the complexity of a class because of inheritance, but the cyclomatic complexity of individual methods can be combined with other measures to evaluate the complexity of the class. Although this metric is specifically applicable to the evaluation of Complexity, it also is related to all of the other attributes. The following formula can be used:

$$ASOM = \frac{\left(\sum_{i=1}^{n}(SOG(i))^2\right)}{NS} \qquad \text{Eq. (3.3)}$$

Where:

ASOM	≡	Average Services Operation Complexity
SOG	≡	Services Operation Granularity (granularity refers to the scope of functionality exposed by a service or component)
NS	≡	No. of Services Domain (NS >0)

Maintainability Index (MI): The MI was computed for the entire system of both approaches since it is not computed on the method or class level.

Depth of Inheritance Tree (DIT): The depth of a class within the inheritance hierarchy is the maximum length from the class node to the root of the tree and is measured by the number of ancestor classes. The deeper a class is within the hierarchy, the greater the number of methods it is likely to inherit making it more complex to predict its behavior. Deeper trees constitute greater design complexity, since more methods and classes are involved, but the greater the potential for reuse of inherited methods.

Impact on quality: High complexity of interactions between the elements of a design unit can lead to decreased understandability and therefore increased fault-proneness. Also, testing such design units is more difficult.

3.4.2.4 Cohesion Metrics:

Cohesion is the degree to which methods within a class are related to one another and work together to provide well-bounded behaviour. Effective object-oriented designs maximize cohesion since it promotes encapsulation. The third class metrics investigates cohesion [68 - 90]. Cohesion metrics measure the relationships among the elements within a single module.

Lack of Cohesion of Methods (LCOM): Lack of Cohesion (LCOM) measures the dissimilarity of methods in a class by instance variable or attributes. It is defined in terms of the number of pairs of class methods that use common class attributes and the number of pairs of class methods that do not use common class attributes. A highly cohesive module should stand alone; high cohesion indicates good class subdivision. Lack of cohesion or low cohesion increases complexity, thereby increasing the likelihood of errors during the development process. High cohesion implies simplicity and high reusability. High cohesion indicates good class subdivision. Lack of cohesion or low cohesion increases complexity, thereby increasing the likelihood of errors during the development process. Classes with low cohesion could probably be subdivided into two or more subclasses with increased cohesion. the following formula can be used:

$$LCOM\ (C) = \frac{NOM - \sum_{a \in C} NOMAF}{NOM - 1} \qquad \text{Eq. (3.4)}$$

Where:

NOM ≡ the total number of methods in the class

NOMAF ≡ the number of methods that access an attribute a in the class (summation for all the attributes in the class)

Cohesion Among Methods of Class (CAM): This metric computes the relatedness among methods of a class based upon the parameter list of the methods. The metric is computed using the summation of number of different types of method parameters in every method divided by a multiplication of number of different method parameter types in a whole class and the number of methods.

Impact on quality: A low cohesive design element has been assigned many unrelated responsibilities. Consequently, the design element is more difficult to understand and therefore also harder to maintain and reuse. Design elements with low cohesion should be considered for refactoring, for instance, by extracting parts of the functionality to separate classes with clearly defined responsibilities.

3.4.2.5 Inheritance Metrics:

Inheritance is a type of relationship among classes that enables programmers to reuse previously defined objects including variables and operators. Inheritance decreases complexity by reducing the number of operations and operators, but this abstraction of objects can make maintenance and design difficult. The two metrics used to measure the amount of inheritance are the depth and breadth of the inheritance hierarchy.

Depth of Inheritance (DIT)

Number of Children (NOC)

Method Inheritance Factor (MIF): MIF is defined as the ratio of the sum of the

inherited methods in all classes of the system under consideration to the total number of available methods (locally defined plus inherited) for all classes. The following formula can be used:

$$MI\ F = \frac{\sum_{i=1}^{TC} Mi(Ci)}{\sum_{i=1}^{TC} Ma(Ci)} \qquad \text{Eq. (3.5)}$$

Where:

Ma (Ci) = Md (Ci) + Mi (Ci)

Attribute Inheritance Factor (AIF): AIF is defined as the ratio of the sum of inherited attributes in all classes of the system under consideration to the total number of available attributes (locally defined plus inherited) for all classes. The following formula can be used:

$$AIF = \frac{\sum_{i=1}^{TC} Ai(Ci)}{\sum_{i=1}^{TC} Aa(Ci)} \qquad \text{Eq. (3.6)}$$

Where:

Aa (Ci) = Ad (Ci)+Ai (Ci)

It is defined analogous to *MIF*.

MOOD- Java Binding:

Ai (Ci) ≡ number of inherited attributes

Ad (Ci) ≡ number of defined attributes with any access

3.4.2.6 Polymorphism Metrics:

Polymorphism Factor (PF): PF is defined as the ratio of the actual number of possible different polymorphic situation for class Ci to the maximum number of possible distinct polymorphic situations for class Ci. The following formula can be used:

3.4.2.7 Encapsulation Metrics:

Method Hiding Factor (MHF): MHF is defined as the ratio of the sum of the invisibilities of all methods defined in all classes to the total number of methods defined in the system under consideration. The invisibility of a method is the percentage of the total classes from which this method is not visible. The following formula can be used:

$$MHF = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{Md(Ci)} (1 - V(Mmi))}{\sum_{i=1}^{TC} Md(Ci)}$$

Where:

$$V(Mmi) = \frac{\sum_{j=1}^{TC} is\_visible(Mmi, Cj)}{TC-1} \qquad \textbf{Eq. (3.7)}$$

And:

$$is\_visible(Mmi, Cj) = \begin{cases} 1 & iff\ j \neq i\ and\ Cj\ may \\ & call\ Mmi \\ 0 & otherwise \end{cases}$$

Attribute Hiding Factor (AHF): AHF is defined as the ratio of the sum of the invisibilities of all attributes defined in all classes to the total number of attributes defined in the system under consideration. The following formula can be used:

$$AHF = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{Ad(Ci)} (1 - V(Ami))}{\sum_{i=1}^{TC} Ad(Ci)}$$

Where:

$$V(Mmi) = \frac{\sum_{j=1}^{TC} is\_visible(Ami, Cj)}{TC-1} \qquad \textbf{Eq. (3.8)}$$

And:

$$is\_visible(Ami, Cj) = \begin{cases} 1 & iff\ j \neq i\ and\ Cj\ may \\ & reference\ Ami \\ 0 & otherwise \end{cases}$$

3.5 Summary

The most challenging aspects of building SOA applications are quality. Despite the increasing importance of service-oriented systems and numerous publications on QoS, this concept still remains rather vaguely defined. It is due to complexity, multi-dimensionality, and multi-sided and context-dependent nature of this concept. The QoS may be considered from a service owner's, requestor's, designer's, network's and other perspectives. Quality assurance should be a central interest from the start, when

developing a service-oriented system. Beginning quality assurance late in the cycle can be costly. Mostly a blend of corresponding quality assurance strategies will be required to attain satisfactory level of quality assurance in service-oriented environments. No single assurance framework is enough until now. Present assurance practices are effective underneath service level.

One of the most important quality models is the quality model presented by McCall et al. [52]. The second basic quality model is the quality model presented by Boehm et al. [53]. Also, the International Organization for Standardization (ISO) [56] issued ISO/IEC 9126 which is currently one of the most widespread quality standards. We have noticed many similarities between the quality models but there are some differences between them. However, it is not a simple task to precisely define the perceived quality, because this quality is associated with subjective estimates depending on the requestor's expectations, past experience and preferences that in turn can be influenced even by the present fashion trends. On the other hand, this concept is very important because the lack of common understanding of QoS is a serious obstacle to direct efforts of all stakeholders of service-oriented systems under development towards a particular common cause. Software metrics are a means of steering development processes by assessing the quality of a software product and finding imbalances in the code base. Metrics have always been used to define the perceived quality and to help guide managers with decisions about their organizations. Several metrics are used such as Size Metrics, Coupling Metrics, Complexity Metrics, Cohesion Metrics, Inheritance Metrics, Polymorphism Metrics and Encapsulation Metrics.

CHAPTER 4: DESIGN DEFECTS

4.1 Introduction

The core of every software system is its architecture. Designing software architecture is a demanding task requiring much expertise and knowledge of different design alternatives. Traditional software engineering attempts to find solutions to problems in a variety of areas, such as testing, software design, requirements engineering, etc. Design patterns are used in software development to provide reusable and documented solutions to common design problems.

Software defects play a major role in determining the quality of a product. Defects occur in any of the phases i.e., requirement phase, design phase, development phase, implementation phase etc. Defects occurrences can be quantified by measuring the defect density and comparing it against the requirements and Service Level Agreement (SLA) contracts. Software faults or defects usually come under the quality factors such as correctness, reliability etc. They have invariable effect and are interdependent directly or indirectly on other quality factors like maintainability, availability, performance, cost/benefit etc. Businesses spend a lot of money trying to fix defects as it affects their outcome. Failure and a fault are considered as defects.

- Failure can be defined as the inability of a system or component to perform its required functions within specified performance requirements and is an observable behavioural deviation from the user requirement or product specification.
- Fault can be defined as an incorrect step, process, or data definition in a computer program which can cause certain failures.

Software design defects will exist as long as software itself exists and is proven to be true in all aspects. Businesses tend to have a defect management system to tackle the problem of solving it or providing information to the users/suppliers regarding the issues. There are products available commercially to manage defects to help businesses.

The key requirements of a useful defect management system are:

- A continual improvement of defect management system until the software / application is retired
- User and role-based allocation of issues/defects
- Defect identification and segregation (multiple products supported)
- Defect cross-referencing – preferably across segregation
- Root cause analysis (incomplete/inaccurate requirements, coding error, unit testing, system testing) [33].

Architectural design decisions determine the ability of the system to meet functional and quality attribute requirements. In the architecture evaluation, the architecture should be analyzed to reveal its strengths and weaknesses, while eliciting any risks. This chapter starts with an introduction to the software defects and explores what makes defects and defects in system development life cycle, followed by a detailed description of design defects in section 4.3. Furthermore, the impact of defects and prioritization depending on the nature of the defect is explored. Additionally, section 4.4 summarizes design attributes and their definitions, whereas section 4.5 discusses defect detection categories and strategies. Finally, section 4.6 presents a summary of the chapter.

4.2 Defects in System Development Life Cycle

In fact, evolving business needs as a result of dynamic and adaptive strategies and market requirements have created a critical demand for defect-free software systems and services. The adoption of service architecture distributed systems, often spanning several geographical locations and cultural differences, has increased the pressure on business analysts, systems architects and developers to quickly deliver usable systems in a cost-efficient manner that serve customers' needs. Such pressure has more than often led to compromise in software quality, particularly in terms of reliability and correctness, as a result of defects leakage from the early stages of projects' life cycles see figure 4.1.

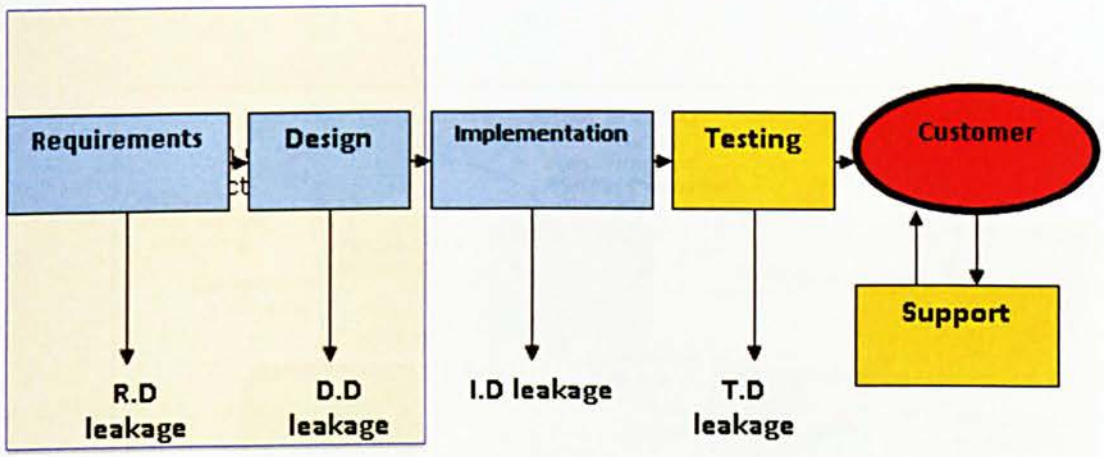

Figure 4.1: Defect Leakages in SDLC

Nevertheless, defect prevention is an essential task in any software project, which should be based on an organised problem-solving methodology to identify, analyse and prevent the manifestation of defects. Defect prevention is a continuing process of collecting the defect data, doing root cause analysis, determining and implementing the corrective actions and sharing the findings learned to avoid future defects. The basic part of the defect prevention process should begin with requirement analysis, which translates customer requirements into product specifications without generating more errors. Next, software architecture is formulated; code reviewed and then testing is carried out to observe the defects, followed by defect logging and documentation, see figure 4.2 illustration of defect prevention [91].

The major advantage of early defect prevention, according to the National Institute of Standard Technology (NIST), is that the cost of fixing one bug found in the production stage of software is 15 hours compared to five hours of effort if the same bug was found in the coding stage. Also, according to the Systems Sciences Institute at IBM, the cost to fix a defect realised after product release is four to five times as much as one recognised during design, and up to 100 times more than one realised in the maintenance phases as shown in figure 4.3, it is much cheaper to fix defects at early stages, such as requirement, design than at late stage of testing [91 & 92].

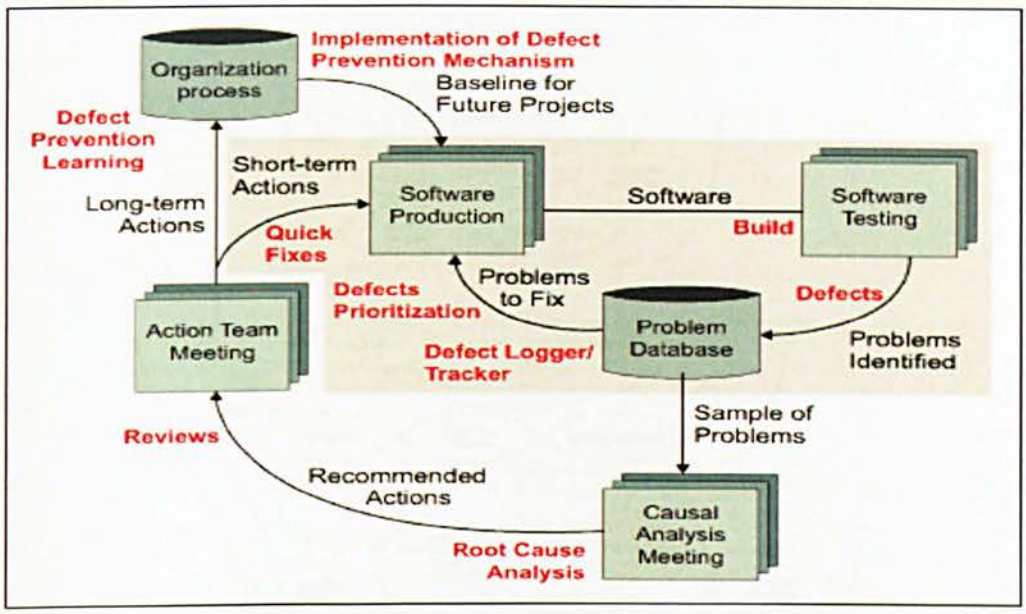

Figure 4.2: Defect Prevention Cycle [91]

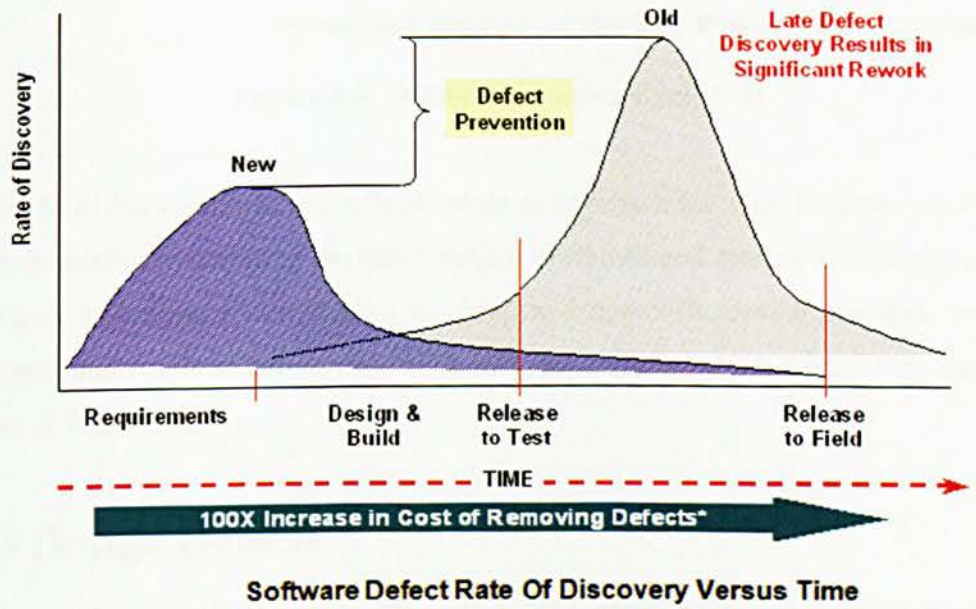

Figure 4.3: Software Defect — Rate of Discovery v/s Time [92]

Figure 4.4 shows a typical waterfall model for the prevention of the defects in software defect life cycle. A better design can prevent the majority of the errors. The aim of the software development process should be to produce high quality products. If followed

rigorously most of the defects can be avoided, also called a linear-sequential life cycle model [93].

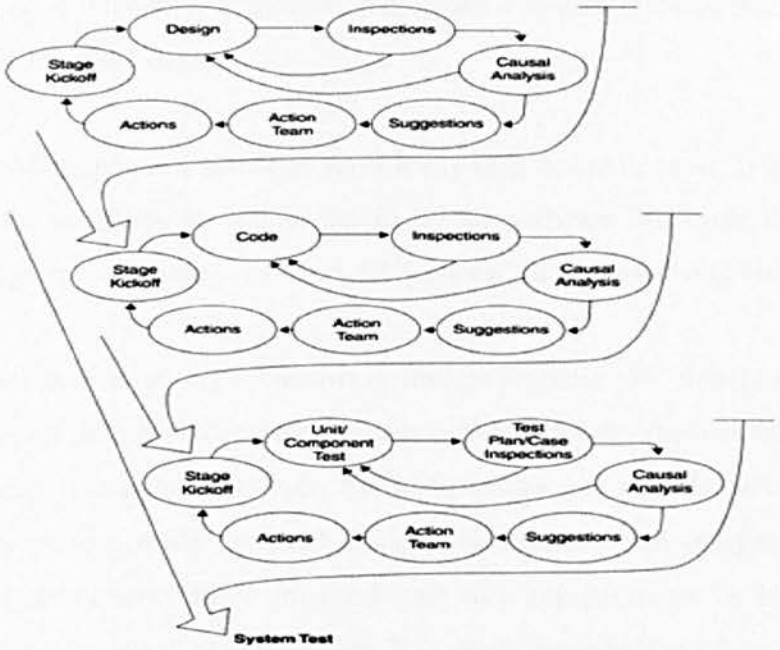

Figure 4.4: Defect Prevention Cycle [93]

To address this need, this research proposes an approach for early assessment of SOA system quality by defining desirable quality attributes and tracing necessary metrics required measuring them. Using the proposed approach, design problems can be detected and resolved before they work into the implemented system where they are more difficult to resolve.

4.3 Design Defects

Software design defects can be defined as "Imperfections in software development processes that would cause software to fail to meet the desired expectations". Defect prevention (DP) is a process of improving quality whose purpose is to identify the common causes of defects, and change the relevant process(es) to prevent that type of defect from recurring.

4.3.1 Defect Identification

Design defects, also called design anomalies, refer to design situations that adversely affect the development of software. Design defects are unlikely to cause failures directly, but may do it indirectly. In general, they make a system difficult to change, which may in turn introduce bugs.

Defects are found by preplanned activities specifically intended to uncover defects. In general, defects are identified at various stages of the software life cycle through activities like design review, code inspection, GUI review, function and unit testing.

Design defects are bad solutions to recurring design problems in object-oriented programming. Design defects occur when system components, interactions between system components, interactions between the components and outside software / hardware, or users are incorrectly designed. Reliable systems are often designed with the possibility of component failure in mind, and with repercussions in place to considerably reduce the odds of system failure. It is worth contemplating how totally engrained the discipline of dependable system design is, outside software engineering.

Developing code free of defects is a major concern for the object-oriented software community. Once defects are identified they are then classified using first level of Orthogonal Defect Classification. Sorting and classifying design defects is complex because of the multiple points of view available.

4.3.2 Defects Classification

Orthogonal Defect Classification (ODC) is the most prevailing technique for identifying defects, wherein defects are grouped into types rather than considered independently. ODC classifies defects at two different points in time:

- When the defect was first detected — opener section
- When the defect was fixed — closer section.

ODC methodology classifies each defect into orthogonal (mutually exclusive) attributes, some technical and some managerial. These attributes provide all the information to be able to shift through the enormous volume of data and arrive at patterns on which root-cause analysis can be done. This coupled with good action planning and tracking, can achieve a high degree of defect reduction and cross learning. For small and medium projects, in order to save time and effort the defects can be classified up to first level of ODC while critical projects, typically large projects, need the defects to be classified deeply in order to analyse and understand.

Basili et al. [90] classified design defects according User Interface (UI) to: Omission, Incorrect Fact, Inconsistency, Ambiguity and Extraneous Information. Whereas, Gueheneuc [94] classified design defects as: Within classes (intra-class), Among classes (inter-classes) and Semantic nature (behavioural).

4.3.3 Design Defects Categories

Based on the structure of the defect, the design defects are classified into the following categories [95]:

- **Interface Capability:** This means a wrong design of the interface, so that it does not give what is expected of it.
- **Interface Specification:** This means a wrong specification of the interface, such that the parameters involved cannot deliver all the information needed to provide the anticipated functionality. The specification of an interface is wrong, so that the parameters involved cannot transfer all of the information required for providing the intended functionality.
- **Interface Description:** This means an incomplete or misleading description of the non-formal parts of the interface. The description of a variable or class attribute or data structure invariant is an (internal) interface as well.
- **Missing Design:** A certain requirement is not covered in the design Boehm observed in 1987 that, "This insight has been a major driver in focusing industrial software practice on thorough requirements analysis and design, on early verification and validation, and on up-front prototyping and simulation to avoid costly downstream fixes."

Also, design defects may be defined as those that are caused by algorithm and processing control, logic and sequence data. In addition, specifically to SOA, such defects may be as a result of module interface description and/or external interface. Such defects may be the result of wrong system component design, overlooked relations between system components, failure for proper analysis description relations between external and internal systems.

Design faults that adversely affect that development process of software are called design defects or anomalies. These can instigate direct or indirect failures that make any changes to a software system difficult and may produce a number of bugs [42]. Code smells and anti-patterns are commonly mentioned in previous studies and literature [41 & 43]. The main purpose of introducing different types of defects is to facilitate their detection and present an amendments process. There are different sets of symptoms that have been defined as common design defects, such as code smells [43].

A "code smell" is any symptom in the source code of a program that possibly indicates a deeper problem. Often the deeper problem hinted by a code smell can be uncovered when the code is subjected to a short feedback cycle, where it is re-factored in small, controlled steps, and the resulting design is examined to see if there are any further code smells that indicate the need of more refactoring. From the point of view of a programmer charged with performing refactoring, code smells are heuristics to indicate when to re-factor, and what specific refactoring techniques to use. Thus, a code smell is a driver for refactoring. The term appears to have been coined by Kent Beck on Wards Wiki in the late 1990s. Usage of the term increased after it was featured in refactoring improving the design of existing code [93]. Code smell is also a term used by agile programmers. Determining what is and is not a code smell is often a subjective judgment, and will often vary by language, developer and development methodology as shown in figure 4.5.

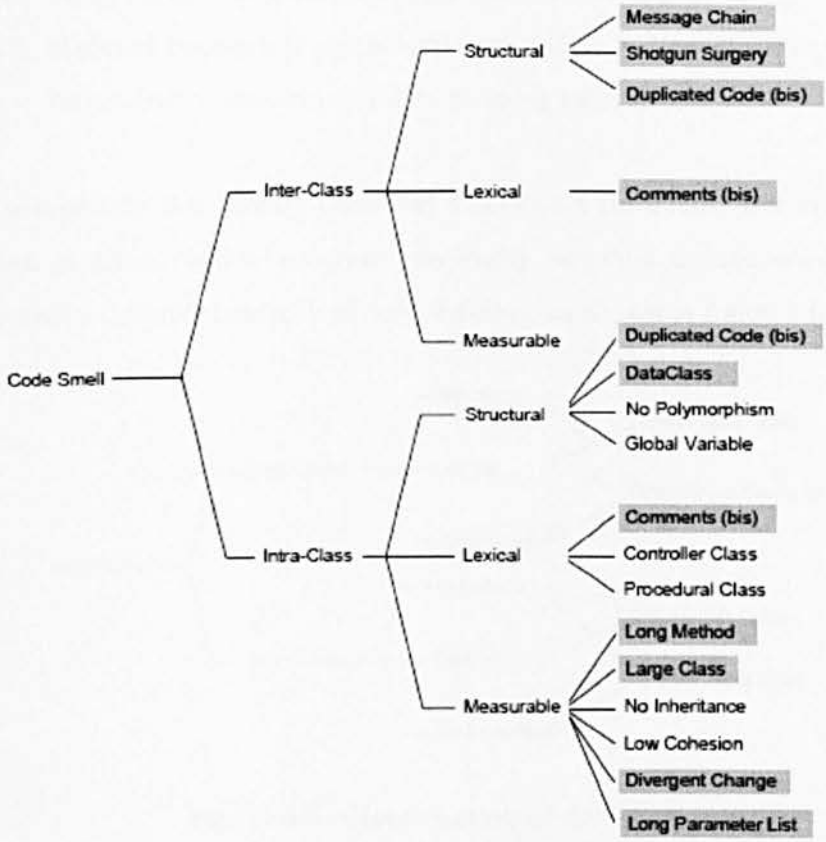

Figure 4.5: Classification of Code Smell [94]

Intra-classes: defects related to the internal structure of a class, whereas Inter-classes: defects related to the external structure of the classes (public interface) and their relations (inheritance, association, etc.). There are tools, such as Check style, PMD and Find Bugs for Java, to automatically check for certain kinds of code smells as follow:

1. **Duplicated code:** Identical or very similar code exists in more than one location.
2. **Long method:** A method function or procedure that has grown too large.
3. **Lazy class:** A class that does too little, or has a very small number of methods; this also can be configured in our framework.
4. **Long parameter list:** The more parameters a method has, the more complex it is. This point also can be configured in the framework.
5. **Conditional complexity:** Large conditional logic blocks, particularly blocks that tend to grow larger or change significantly over time.

6. **Dead code:** The dead code is the code that never called.
7. **Refused bequest:** If you inherit from a class, but never use any of the inherited functionality, should you really be using inheritance.

An anti-pattern is a literary form that describes a commonly occurring solution to a design problem, which generates decidedly negative consequences. Anti-patterns represent a different category of design defects as shown in figure 4.6.

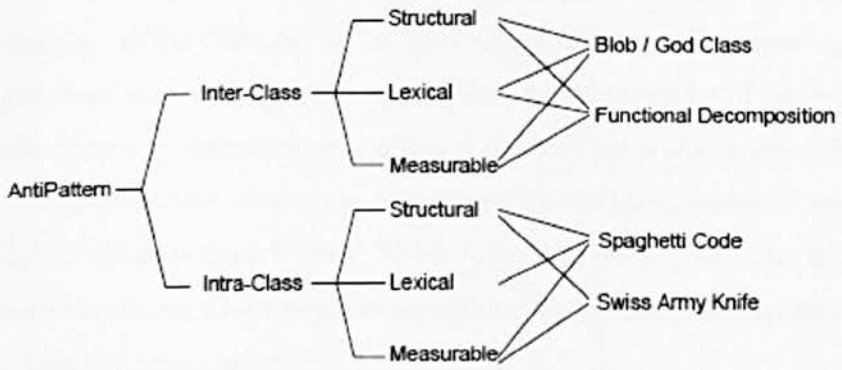

Figure 4.6: Classification of Anti-Patterns [95]

1-Blob: Single class with a large number of attributes or operations, or both, usually indicates the presence of the Blob.

2-Functional Decomposition: Also known as no object-oriented anti-pattern. This anti-pattern is the result of experienced, non-object-oriented developers who design and implement an application in an object-oriented language. When developers are comfortable with a "main" routine that calls numerous sub-routines, they may tend to make every sub-routine a class, ignoring class hierarchy altogether (and pretty much ignoring object orientation entirely). The resulting code resembles a structural language such as C or FORTRAN in class structure. It can be incredibly complex, as smart procedural developers devise very clever ways to replicate their time-tested methods in an object-oriented architecture.

3-Spaghetti Code: Spaghetti Code appears as a program or system that contains very little software structure. Coding and progressive extensions compromise the software structure to such an extent that the structure lacks clarity, even to the original developer, if he or she is away from the software for any length of time. If developed using an object-oriented language, the software may include a small

number of objects that contain methods with very large implementations that invoke a single, multi-stage process flow. Furthermore, the object methods are invoked in a very predictable manner, and there is a negligible degree of dynamic interaction between the objects in the system. The system is very difficult to maintain and extend, and there is no opportunity to reuse the objects and modules in other similar systems.

4-Swiss Army Knife: A Swiss Army Knife, also known as Kitchen Sink, is an excessively complex class interface. The designer attempts to provide for all possible uses of the class. He or she adds a large number of interface signatures in a futile attempt to meet all possible needs. Real-world examples of Swiss Army Knife include dozens to thousands of method signatures for a single class. The designer may not have a clear abstraction or purpose for the class, which is represented by the lack of focus in the interface. Swiss Army Knives are prevalent in commercial software interfaces, where vendors are attempting to make their products applicable to all possible applications.

5-Poltergeists: Poltergeists are classes with limited responsibilities and roles to play in the system; therefore, their effective life cycle is quite brief. Poltergeists clutter software designs, creating unnecessary abstractions; they are excessively complex, hard to understand, and hard to maintain. This anti-pattern is typical in cases where designers familiar with process modelling but new to object-oriented design define architectures. In this anti-pattern, it is possible to identify one or more ghostlike apparition classes that appear only briefly to initiate some action in another more permanent class. The Poltergeist anti-pattern is usually intentional on the part of some greenhorn architect who doesn't really understand the object-oriented concept.

Nevertheless, quality managers and project managers need to identify, while at the same time engage in, a strategy for defect detection and classification that adds value and benefits the project [65]. This is one of the driving parameters of the project. It is therefore evident that such defects detection and classification should be highly effective and efficient, according to [96], in order to find a balance between the defect cost and customer detected defects.

Developers will understand the source code easily if the defects have been detected and removed. However, it is time-consuming, as it is to some extent a manual process [91].

The amount of resources available may not cover all the type of defects, as there are known and unknown defects. Thus, several automated detection techniques have been developed to handle this enormous amount of activity [[43, 94, 97 - 99].

Design defect detection and classification therefore should be done early into the project to ensure the organization benefits from a cost perspective, as such defects are costly to resolve, especially if noticed later into the project [91].

On the other hand, various and extensive research work and solutions have been done, especially in the recent past, which focus on object oriented software design in the development cycle. These solutions use tools for the detection and classification of design defects that demonstrate high levels of precision.

Nevertheless, all these tools are based on the ability to detect defects and classify them without diligently factoring the cost of addressing such defects, which calls for a proper classification framework, and correct information attachment on the reported defects. It is quite necessary to have a solution developed from the perspective of business requirements rather than system perspective. Table 4.1 shows an example of SOA Design Defects and their definitions.

Table 4.1: Design Defects

SOA Design Defects	Definition
Algorithmic and Processing Defects	These occur when the processing steps in the algorithm as described by the pseudo code are incorrect. In the latter case a step may be missing or a step may be duplicated. In the case of algorithm reuse, a designer may have selected an inappropriate algorithm for this problem (it may not work for all cases)
Control, Logic, and Sequence Defects	Control defects occur when logic flow in the pseudo code is not correct. Logic defects usually relate to incorrect use of logic operators, such as less than <, greater than >, etc. These may be used incorrectly in a Boolean expression controlling a branching instruction.
Omission	Necessary information about the system has been omitted from the software artifact.
Incorrect Fact	Some of the information in the software artifact contradicts with the information in the requirements document or the general domain knowledge for the usage of the software.
Inconsistency	The information within one part of the software artifact is inconsistent with other information in the software artifact and such types of user design could also lead to defect.
Ambiguity	Information within the software artifact is ambiguous, i.e. any of a number of interpretations may be derived that should not be the prerogative of the developer doing the implementation.
Extraneous Information	Information is provided that is not needed or used can also confuse the user and lead to defects.
Functional Description Defects	The defects in this category include incorrect, missing, and/or unclear design elements. These defects are best detected during a design review.

Intra-class Defects	This category includes any design defect related to the internal structure of a class.
Behavioral Defects	All the design defects related to the application *semantics* belong to this category.
Inter-class Defects	This category encloses any design defect related to the external structure of the classes (their public interface) and their relationships.
Ambiguous Design	Design feature/approach is not clear to the reviewer. Also includes ambiguous use of words or unclear design features.
Ambiguous Requirements	Requirement is not clear to the reviewer. Also includes ambiguous use of words – e.g. Like, such as, may be, could be, might etc.
Superfluous	Some information of the SRS document is not relevant to the problem being solved or will not contribute to the solution.
Not-conforming to standards	Some items in the requirement are written in a way not conforming to the standards determined by quality assurance representatives.
Not-implementable	Some requirements are not implementable due to system constraints, human resources, budget, or technology limitations.
Risk-prone	Some requirements are risk prone due to unstable requirements or requirements with high interdependence.

4.4 Design Attributes

Perepletchikov et al. [76] provided a comparative study on the impact of object orientation and service orientation on the structural attributes of size, complexity, coupling and cohesion.

Recently, Shaik et al. [81] studied design components that were exclusive and defined the architecture of an object oriented design and listed the key terms in object oriented development environment: Class, Object, Method, Message Instantiation, Inheritance,

Polymorphism, Encapsulation, Cohesion, Coupling, Design Size, Hierarchies, Abstraction and Complexity.

In 2011, Yaser and Suleiman [50] assessed software quality attributes of Service-Oriented Software Development Paradigms using four SOA design attribute: Size, Complexity, Coupling and Cohesion. Table 4.2 shows a summary of Design Attributes and their definitions.

Table 4.2: Design Attributes

Design Attributes	Definition
Class	A set of objects that share a common structure and common behaviour manifested by a set of methods; the set serves as a template from which objects can be instantiated.
Object	An instantiation of some class which is able to save a state (information) and which offers a number of operations to examine or affect this state.
Method	An operation upon an object, defined as part of the declaration of a class. Methods are operations but not all operations are actual methods declared for a specific class.
Message	A request that an object makes of another object to perform an operation.
Design Size	Design size measures the size of design elements, typically by counting the elements contained within the design. For example, the number of operations in a class, the number of classes in a package, and so on.
Complexity	Complexity measures the degree of connectivity between elements of a design unit. Whereas size counts the elements in a design unit, and coupling the relationships/dependencies leaving the design unit boundary, complexity is concerned with the relationships/dependencies between the elements in the design unit. For instance, counting the number method invocations among the methods within one class can be considered a measure of class complexity, or the number of transitions between the states in a state diagram.

Coupling	Coupling is the degree to which the elements in a design are connected.
Cohesion	Cohesion is the degree to which the elements in a design unit (package, class etc.) are logically related, or "belong together". As such, cohesion is a semantic concept.
Inheritance	A relationship among classes wherein one class shares methods defined in one (for single inheritance) or more (for multiple inheritance) other classes.
Polymorphism	The ability of an object to interpret a message differently at execution depending upon the super class of the calling object.
Encapsulation	The process of bundling together the elements of an abstraction that constitute its structure and behaviour.
Hierarchies	Hierarchies are used to represent different generalization-specialization concepts in a design.
Abstraction	A measure of the generalization specialization aspect of the design.
Instantiation	The process of creating an instance of the object and binding or adding the specific data.

4.5 Defect Detection

Software defect prevention is an important part of the software development. The quality, reliability and cost of the software product heavily depend on the software defect detection and prevention process. In the development of software product 40% or more of the project time is spent on defect detection activities. Defects are found by pre-planned activities specifically intended to uncover defects. In general, defects are identified at various stages of software life cycle through activities like design review, code inspection, GUI (graphical user interface) review, function and unit testing. Once defects are identified they are then classified using the first level of Orthogonal Defect Classification.

Software reviews have been extensively studied. However, very little information on the detected defect types was provided in the most recent review articles. Different

techniques, frameworks and strategies focused on detecting and fixing design defects in software have been presented in several studies [100 - 103]. However, most of the previous work has focused on solving design defect problems in traditional applications and monolithic architectures. On the other hand, the current study is focusing on detecting defects in distributed component-based applications, particularly those based on SOA paradigm.

4.5.1 Defect Detection Categories

Defect classification is the most prevailing technique for identifying defects wherein defects are grouped into categories rather than considered independently. Defects detection research work can be classified into three broad categories:

- Visual detection and inspection
- Rules-driven detection-correction
- A combination of detection and correction techniques.

The first category, visual detection and inspection, is based on the available visualization design environments combined with human ability to analyze, conceptualize and incorporate previous knowledge and design expertise. The ability to inspect complex contextual information is fundamental for design defect detection. Approaches for visual design detection have been proposed in literature. A pattern-based framework for the detection of software anomalies by representing potential defects with different colours was proposed by Kothari et al [104]. Another approach presented by Dhambri [105] is based on semi-automatic detecting of design anomalies and defects by combining automatic defects symptoms detection with human analysis. The main issue with the visualization approach is the fact that it is hard to evaluate for complex large-scale systems.

The second category, rules-driven detection-correction, is based on a set of predefined rules and quality metrics. Such an approach is clearly based on a well-defined list of rules and metrics, as proposed by Marinescu [99], for detecting design defects in object-oriented design at system, sub-system, class and method levels. Other works focused on the use of metrics to improve the accuracy of detection and for frameworks evaluation, as proposed in [104], where the concept of multi-metrics, n-tuples of

metrics expressing a quality criterion, has been presented and discussed. The rules, however, require defining threshold values for the metrics, which has been addressed in [106] where defect detection is expressed as "fuzzy rules with fuzzy labels for metrics".

Other rule-driven approaches have adopted an abstract rule language to describe design defects symptoms, such as the DÉCOR approach. This involves describing classes, structures etc and their roles, which are then mapped to detection algorithms. The approach also adopts a heuristic approximation of the threshold values for the metrics [66] DÉCOR was further extended in [97] to sorting defects and to support uncertainty, which was, according to Bayesian belief, networks that implement the detection rules of DÉCOR.

The third category of work is based on implicit detection of defects, ie they are not detected explicitly because the approaches generate a refactoring strategy, which fixes detected defects, first by detecting elements that can be changed to improve the quality criterion. A refactoring approach based on problem optimisation was proposed in [107], where up to 12 metrics were used to measure the impact of refactoring, including simple ones such as moving methods between classes. Overall the aim of the optimisation is to find out the sequence that maximises a function reflecting the variations of metrics [108].

4.5.2 Defect Detection Strategies

4.5.2.1 Walkthrough and Visual Inspection

A walkthrough involves a statement of objectives for the entire process, the software product, and any regulations, standards or guidelines. The process is considered successful when the entire software has been examined, and recommendations have been addressed [43].

A prominent feature of the walkthrough strategy in design is that it allows the designers to obtain early substantiation of the design decisions related to software. The scope of a walkthrough covers design of the GUI, treatment of content and elements of the

software functionality. Walkthroughs are important to both the designer and the customer, in that they provide a way to access and identify whether the design addresses the project's goal and meets the requirements.

An effective walkthrough has to include specific components, in an effort to relay the design specifics to the customer. The developer guides members of the development team and other interested parties through a segment of the design. The aim of a walkthrough is to get a valid feedback from the client or peers, i.e. other developers. Usually, the team comments on standards, errors or violations in the development process [109].

Some aspects of walkthroughs pose potential drawbacks to the process. First, the designer has to prepare for the meeting. This involves coordinating the effort and time of each participant and making sure that their personal work plans are synchronised with the project's schedule [110]. Inadequate individual preparation may result in poor review, or misconception of the principles applied in the design. Another aspect of a walkthrough that may make it ineffective is the selection of the right participants. It is important to invite the participants with relevant knowledge background and skills to make the exercise meaningful for all. Inviting the right participants ensures that the walkthrough adds value and quality to the product and not to the participants learning [111].

The improvement of the project quality is of the utmost importance. This assists in increasing team morale and hence enhances the development process. For a walkthrough to be successful and systematic, at least two members have to be involved. The walkthrough leader serves as the author or recorder. A walkthrough member should not hold a managerial post over other members.

4.5.2.2 Object-Oriented Defect Detection

Design defects originate from poor design choices. They degrade quality of the designs; therefore, they present opportunities for quality improvement. The design defects are defined as wrong solutions to regular problems in object-oriented design. Basically, they come from UML class diagrams that encompass problems at different levels of complexity. Defects in object-oriented applications arise as a result of poor design

choices which cause degrading effects on the [110]. Various tools and methods have been developed to aid in error detection and correction during software development. However, due to non-specification of design defects, there exist a few appropriate methods for detection.

In object-oriented designs, defects are defined as wrong solutions to recurring problems. Problems may occur at different levels of design, ranging from the architectural level, anti-patterns, to the low level, such as code smells. A good example of a common defect in object-oriented applications is the "spaghetti code", which involves unstructured classes, thus declaring long methods with no parameters [112].

Defect detection and correction in object-oriented programming is done early in software development to reduce development costs for subsequent steps. Designs that are free from defects are easy to implement. These defect detection procedures may be time and resource consuming. Various approaches have been developed to detect and correct defects in object-oriented designs.

However, design detection has some shortcomings where the design defects are not precisely specified. It only provides a systematic method that can automatically detect and correct the errors. The size of the software applications makes it harder to achieve non-defective design using this methodology. In addition, the object-oriented defect detection can be expensive which is due to the complexities of software designs, hence requiring professionals and experienced designers.

4.5.2.3 DÉCOR Method

DÉCOR stands for Defect Detection for Correction [113]. It is applied to specific high-level design defects, and determines correction algorithms based on defect specification. This method employs four main stages, from analysis of the defect, to detection and correction of defects.

Specification is the first stage in this method. It entails characterization of all the defects based on their characteristics and effect on the system. Taxonomy is established, describing terminology and classification of design defects related to theoretical

descriptions to avoid misinterpretation. From the specification of the system's design goals, defects can be detected by comparing the system's performance to its design goals. A model of the system is created for easy analysis of possible sources of defects.

Detection from the specified areas follows. Techniques and algorithms are defined to detect design defects from the system model previously developed. These techniques are based on semantics, structure and metrics of the system. The system metrics define its size, complexity, coupling and cohesion [113]. Defects in a system are directly proportional to the magnitude description of the system's metrics. Metric values are classified into five different levels: very low, low, medium, high and very high.

Corrections are done sequentially while testing the system to determine proper system functionality. Improvements on the design are made precisely with the intention of matching the systems performance to the intended goals. After the correction of the detected defects in the design, the software is then validated.

Validation involves a series of steps and experiments to evaluate system performance after having corrected the design defects of the system. The previous performance is compared to the current performance and functionality to determine the effect after error correction.

It is essential to specify the design defects in object-oriented programming. This acts as a framework for generating detection algorithms of a system. There exist methods developed to generate detection algorithms automatically, based on specifications written using a domain-specific language [113]. Thus, a framework for the automatic detection and the classification of design defects is proposed in the next chapter.

4.5.3 Defect Prevention Activities

The five general activities of defect prevention are [91]:

1. Software Requirements Analysis

Errors in software requirements and software design documents are more frequent than errors in the source code itself, according to *Computer Finance Magazine* [114].

Defects introduced during the requirements and design phase are not only more probable but also are more severe and more difficult to remove. Front-end errors in requirements and design cannot be found and removed via testing, but instead need pre-test reviews and inspections.

2. Reviews: Self-Review and Peer Review

Self-review is one of the most effective activities in uncovering the defects which may later be discovered by a testing team or directly by a customer. The majority of the software organizations is now making this a part of "coding best practices" and is really increasing their product quality.

3. Defect Logging and Documentation

Effective defect tracking begins with a systematic process. A structured tracking process begins with initially logging the defects, investigating the defects, then providing the structure to resolve them. Defect analysis and reporting offer a powerful means to manage defects and defect depletion trends, hence, costs.

4. Root Cause Analysis and Preventive Measures Determination

After defects are logged and documented, the next step is to analyze them. Generally the designated defect prevention coordinator or development project leader facilitates a meeting to explore root causes. Root cause analysis is the process of finding and eliminating the cause, which would prevent the problem from recurring. Finding the causes and eliminating them are equally important. The cause-and-effect diagram, also known as a fishbone diagram, is a simple graphical technique for sorting and relating factors that contribute to a given situation.

Once the root causes are documented, finding ways to eliminate them requires another round of brainstorming. The object is to determine what changes should be incorporated in the processes so that recurrence of the defects can be minimized.

5. Embedding Procedures into Software Development Process

Implementation is the toughest of all activities of defect prevention. It requires total commitment from the development team and management. A plan of action is made for deployment of the modification of the existing processes or introduction of the new

ones with the consent of management and the team. Monthly status of the team should mention the severe defects and their analyses.

4.6 Related Approaches

Agile methodologies embrace iterations [92]. Teams work together with stakeholders to define quick prototypes, proof of concepts, or other visual means to describe the problem to be solved. The team defines the requirements for the iteration, develops the code, and defines and runs integrated test scripts, and the users verify the results.

The most widely used methodologies based on the agile philosophy are XP and Scrum. These differ in particulars but share the iterative approach described above.

- **XP**: XP stands for extreme programming. It concentrates on the development rather than managerial aspects of software projects. XP was designed so that organizations would be free to adopt all or part of the methodology.
- **Scrum**: In rugby, "scrum" (related to "scrimmage") is the term for a huddled mass of players engaged with each other to get a job done. In software development, the job is to put out a release. Scrum for software development came out of the rapid prototyping community because prototypes wanted a methodology that would support an environment in which the requirements were not only incomplete at the start, but also could change rapidly during development. Unlike XP, Scrum methodology includes both managerial and development processes.

Early feedback on defects in SDLC will fit within the agile methodology as they are communicated to stakeholders, and agreed on the identification and corrections of defects as well as quality assurance before moving to the next stages of the software development lifecycle such as implementation and testing.

From an architectural point of view, the Architecture Defect Detection (ATAM) [93] considers how early architectural decisions define how the system is organized in terms of permanent data management, data communication, data input and output, coarse-

grained modularization and allocation within the organizational structure. Such a system's "back-bone" has been referred to as the System Organization Pattern.

Analyzing architecture early in the development life cycle can help identify significant technical risks and mitigate them at a minimal cost. However, architecture assessment methods, such as the Architecture Trade-off Analysis Method, cannot easily be applied very early for architecture defined only conceptually. In addition, the influence of the System Organization Pattern on the detailed properties of the final system cannot be precisely quantified, which makes applying known architecture analysis methods even more difficult. The Early Architecture Evaluation Method has been developed to assess the System Organization Pattern much earlier than an ATAM-based assessment would be possible, i.e. in the inception phase of the Rational Unified Process. The method defines an architecture evaluation process, at the heart of which is an assessment model based on the Goal-Question-Metric scheme. The method identifies substantial risks posed by the architectural decisions comprising the System Organization Pattern. The method has been evaluated on seven real-life examples of large-scale systems.

4.7 Summary

Software design is one of the most important and key activities in the system development life cycle (SDLC) phase that ensures the quality of software. Different key areas of design are very vital to be taken into consideration while designing software. Software design describes how the software system is decomposed and managed in smaller components. From the studies made by various software development communities, it is evident that most serious failures in software products are due to errors in the requirements and design phases. The detection of design defects is important to improve the quality of software systems, to ease their evolution, and thus to reduce the overall cost of software development. The computation times of the design defects vary with the design defects and the systems. However, the manual detection of design defects is tedious and time-consuming.

Defect prevention is an essential task in any software project. Defect prevention is a continuing process of collecting the defect data, doing root cause analysis, determining and implementing the corrective actions and sharing the findings learned to avoid future

defects. Design defects are bad solutions to recurring design problems in object-oriented programming. Based on the structure of the defect, the design defects are classified into Interface Capability, Interface Specification, Interface Description and Missing Design. Software defect prevention is an important part of the software development. The quality, reliability and cost of the software product heavily depend on the software defect detection and prevention process. The main Defect Detection Strategies are: Walkthrough and Visual Inspection, Object-oriented Defect Detection and DÉCOR Method. Finally, defect prevention is not an individual exercise but a team effort. The software development team should be striving to improve its process by identifying defects early, minimizing resolution time and therefore reducing project costs. There are a number of issues with the discussed approaches. Firstly, they are mainly related to object oriented paradigm and do not tackle service oriented architecture; secondly, they are not fully automated; and finally, they do not link software quality with defect designs.

CHAPTER 5: THE PROPOSED FRAMEWORK

5.1 Introduction

The SOA model has been realized from the need for an interdisciplinary and enhanced service modelling approach. SOA is an emerging architectural style that is instrumental in creating next-generation applications. In SOA software design, there two key principles for the entire process, pattern and anti-patterns. Patterns are guidance steps and best practices used in the design process. Anti-patterns are common design flaws in the process of coming up with a software product [114]. Recognising anti-patterns is a fundamental part of software development; it allows the developer to learn from previous mistakes.

Metric computation is a way of evaluating software's capabilities in terms of size, cohesion or complexity. It is derived from quality goals in the documentation. Metrics in SOA software design involves the evaluation of models and business processes. The detection process involves correct identification of patterns and anti-patterns. This process may be fuzzy or inaccurate, but all possible design flaws can be identified by locating discontinuities in the patterns [115]. Primitive rules may also suggest the presence of anti-patterns. A more accurate detection process can also be undertaken. Defect identification may seem impractical since one needs to identify each defect from the fuzzy analysis. Thus, historically design defect detection is done manually using visual detection and inspection.

This chapter starts with the proposed framework on design defects and software quality assurance (DESQA), the framework description in section 5.2 includes its objectives and assumptions together with a detailed and comprehensive description of the framework. Section 5.3 describes in details the process of using the proposed framework and its formalization. Section 5.4 describes the framework design execution including the potential technologies for its applications. Finally, section 5.5 presents a summary of the chapter.

5.2 Design Defects and Software Quality Assurance Framework DESQA

Defect prevention is an important activity in any software project. In most software organizations, the project team focuses on defect detection and rework. Thus, defect prevention, often becomes a neglected component. It is therefore advisable to make measures that prevent the defect from being introduced into the software products right from early stages of the project. Defect prevention provides the greatest cost and schedule savings over the duration of the application development efforts. Detection of errors in the development life cycle helps to prevent the migration of errors from requirement specification to design and from design into code. The DESQA framework will be applied to UML (Unified Modeling Language) design diagrams, which are generated from the project requirement specifications, in order to minimize design defects leakage into the implementation stage. In addition, the DESQA framework aims to provide a software quality estimation, thus linking design defects to particular software quality factors.

5.2.1 Framework Objective

Developing quality code is a major concern for the software community. Producing bug-free, extensible, and adaptable code is a hard task. It requires skills, experience, and a deep understanding of the structure and behaviour of the software under development. Many studies [17, 20, 53, 71 & 73] addressed the problems of automating the detection and the correction of design defects. The link between design defects, metrics and software quality factors has not been addressed; in addition the impact of design defects on software quality estimation has not been evaluated. Thus, there is a clear need for an integrated framework not only for the identification of the design defects but also for providing software quality estimation. Consequently, the purpose of this study is to propose a framework to automate the detection of design defects based on design patterns and using design constraints, and to use software metrics for measurement of defects and estimation of software quality factors.

5.2.2 Framework Assumptions

The DESQA framework was based on the following assumptions:

- The DESQA framework can be used with Design phase only (Conceptual, Preliminary and Final design phases).
- All SOA design attributes, quality metrics, SOA design defects and SOA quality attributes are well defined and can be measured.

5.2.3 Framework Description

Starting with eliciting the project requirement, and following the requirement specification a detailed design of the system is produced. The design is then converted to some sort of description, for example textual form, which is then analysed by an intelligent parser. The parser has two functions, first it checks against an existing defect database for potential defects. It also checks for potential corrective action using the defects portfolio. Figure 5.1 shows the stages of the DESQA framework as well as the flow between the stages of the framework.

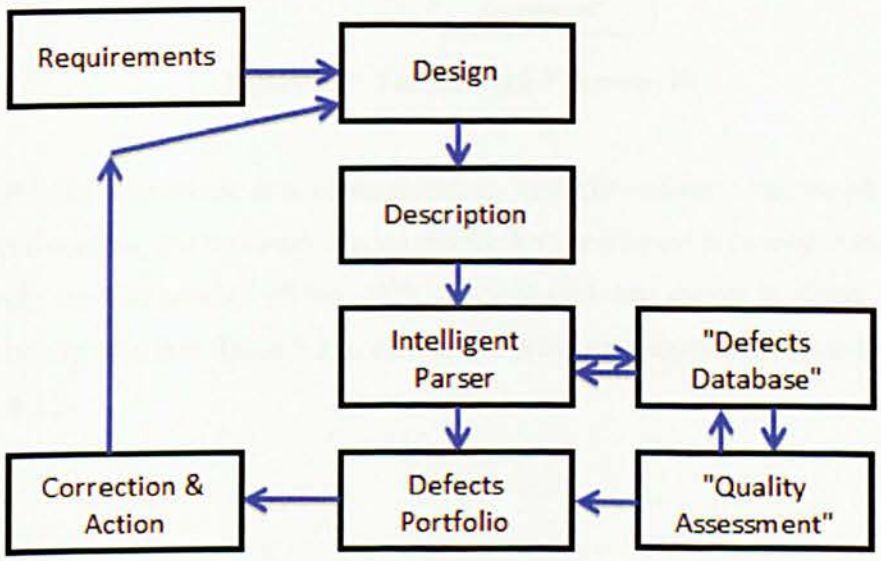

Figure 5.1: The Functionality of the Framework

In the DESQA framework the design is considered in three stages, conceptual design, preliminary design and final design. The defects database is reflex the different stages. Moreover, code generation and intelligent techniques for defect detection and defect portfolio are also used as shown in figure 5.2.

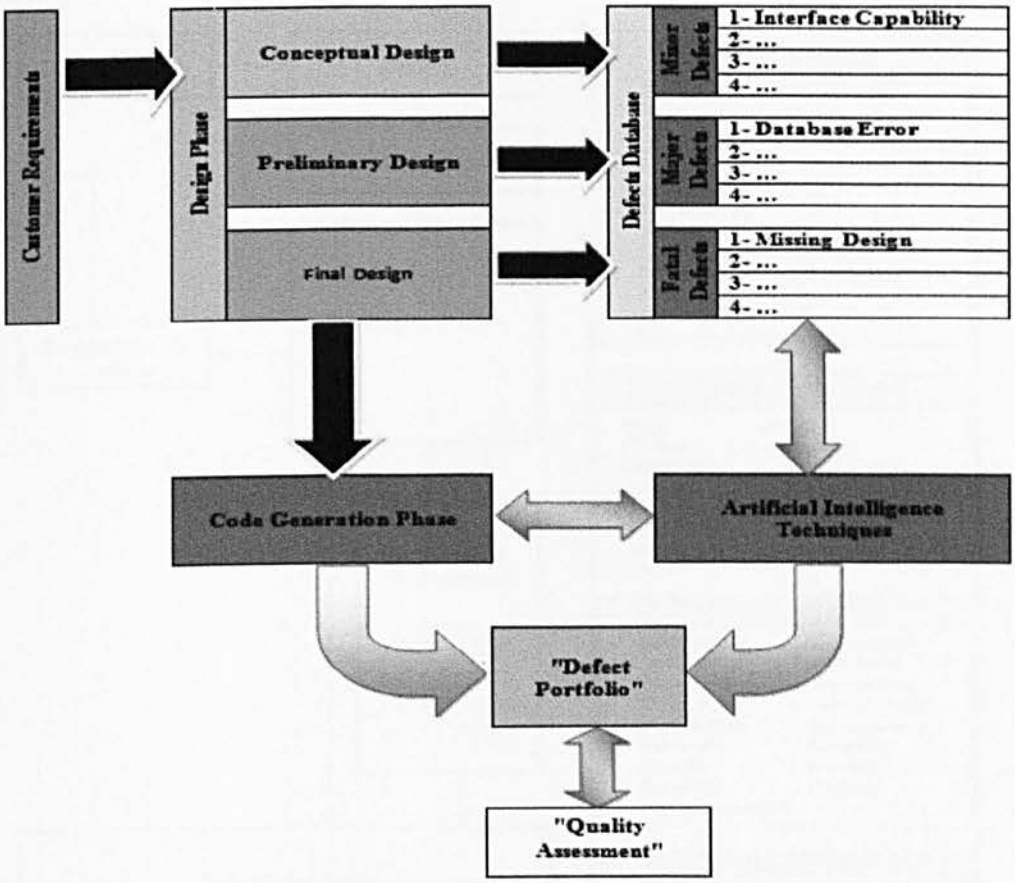

Figure 5.2: The DESQA Framework

The DESQA framework is a comprehensive, multidimensional framework of SOA defects detection. The measures used in this work were adapted primarily from previous research; the components of the DESQA framework are shown in figure 5.3. It is important to note that figure 5.3 integrates the previous diagrams presented in figures 5.1 and 5.2.

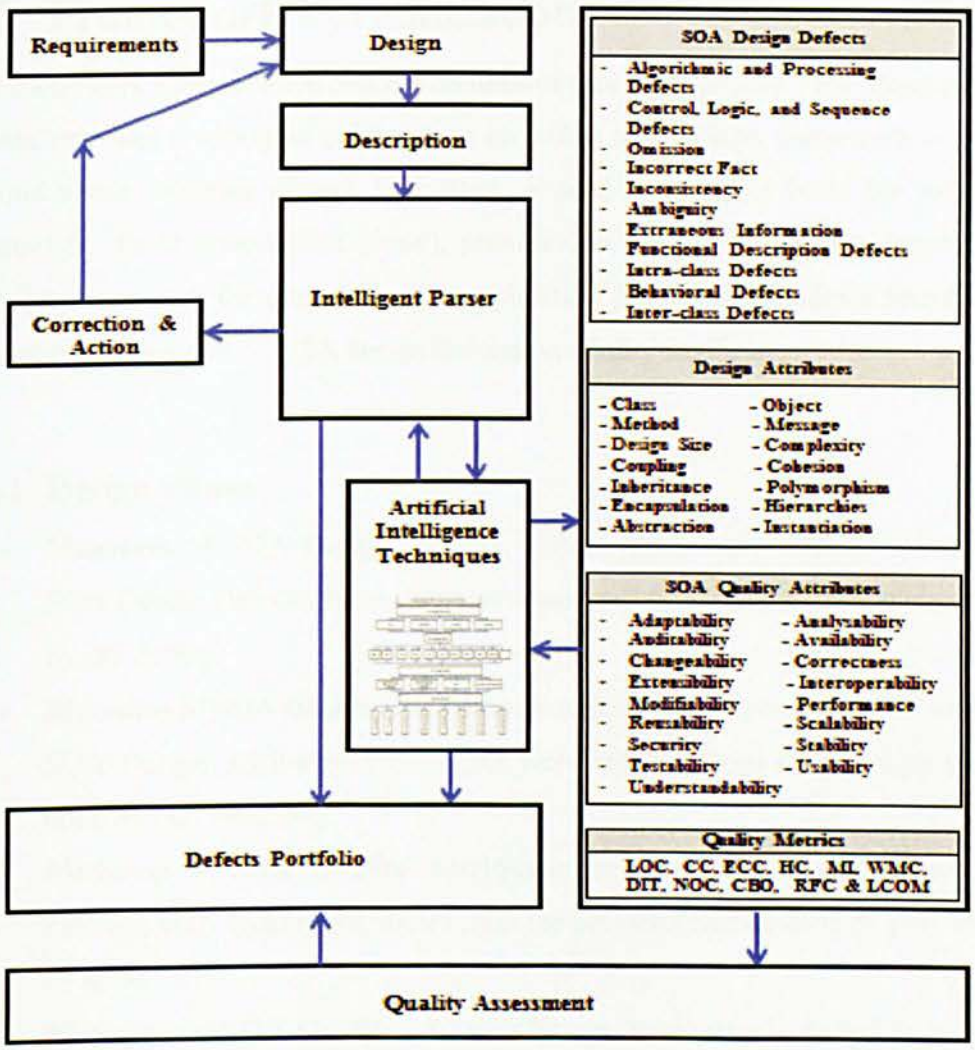

Figure 5.3: The DESQA Framework Components

In reality, however, every study has interpreted and classified quality system metrics conform to its context, the proposed approach consists of eleven items to measure SOA Design Defects, fourteen items to measure SOA Design Attributes, seventeen items were selected to measure SOA Quality Attributes and eleven items to measure SOA Quality Metrics. The user can adapt the number of selected items according to the actual case.

5.3 Framework Formalization

The framework formalization process consists of four main phases. First phase (design phase), provides a survey of publications on which the DESQA framework is based. Second phase (building phase), four steps or parts are used to build the proposed framework. Third (preparation phase), provides six main steps used to prepare the DESQA framework for usage. Finally (application phase), it provides a formula for measuring the impacts of SOA design defects on quality attributes.

5.3.1 Design Phase

- **Measures of SOA Design Defects:** Eleven items were selected to measure SOA Design Defects; these items were selected from the previous studies done by [85 & 90].
- **Measures of SOA Design Attributes:** Fourteen items were selected to measure SOA Design Attributes; these items were selected from the previous studies done by [43, 86 & 90].
- **Measures of SOA Quality Attributes:** Seventeen items were selected to measure SOA Quality Attributes from the previous studies done by [34, 45, 46, 55 & 57 - 61].
- **Measures of SOA Quality Metrics:** Eleven items were selected to measure SOA Quality Metrics from the previous studies done by [43, 55, 59, 71 – 74 & 77 - 81].

5.3.2 Building Phase

The intent of these measures is to measure customer satisfaction by assessing the design defects and its impacts on design quality. The approach tool consists of four parts and their relationship as shown in figure 5.4:

- **Part (I) represents SOA design attributes.**
- Example (but not limited to): size, complexity, coupling, and cohesion.
- **Part (II) represents metrics may be used to measure defects' impact on quality attributes.**

- Example (but not limited to): LOC, CC, ECC, HC, MI, WMC, DIT, NOC, CBO, RFC & LCOM.
- **Part (III) represents SOA design defects.**
- Example (but not limited to): Algorithmic and Processing Defects, Control, Logic, and Sequence Defects, Data Defects and Functional Description Defects.
- **Part (IV) represents SOA quality attributes.**
- Example (but not limited to): Availability, Security, Performance, Modifiability, Scalability, Adaptability, Interoperability and Auditability.

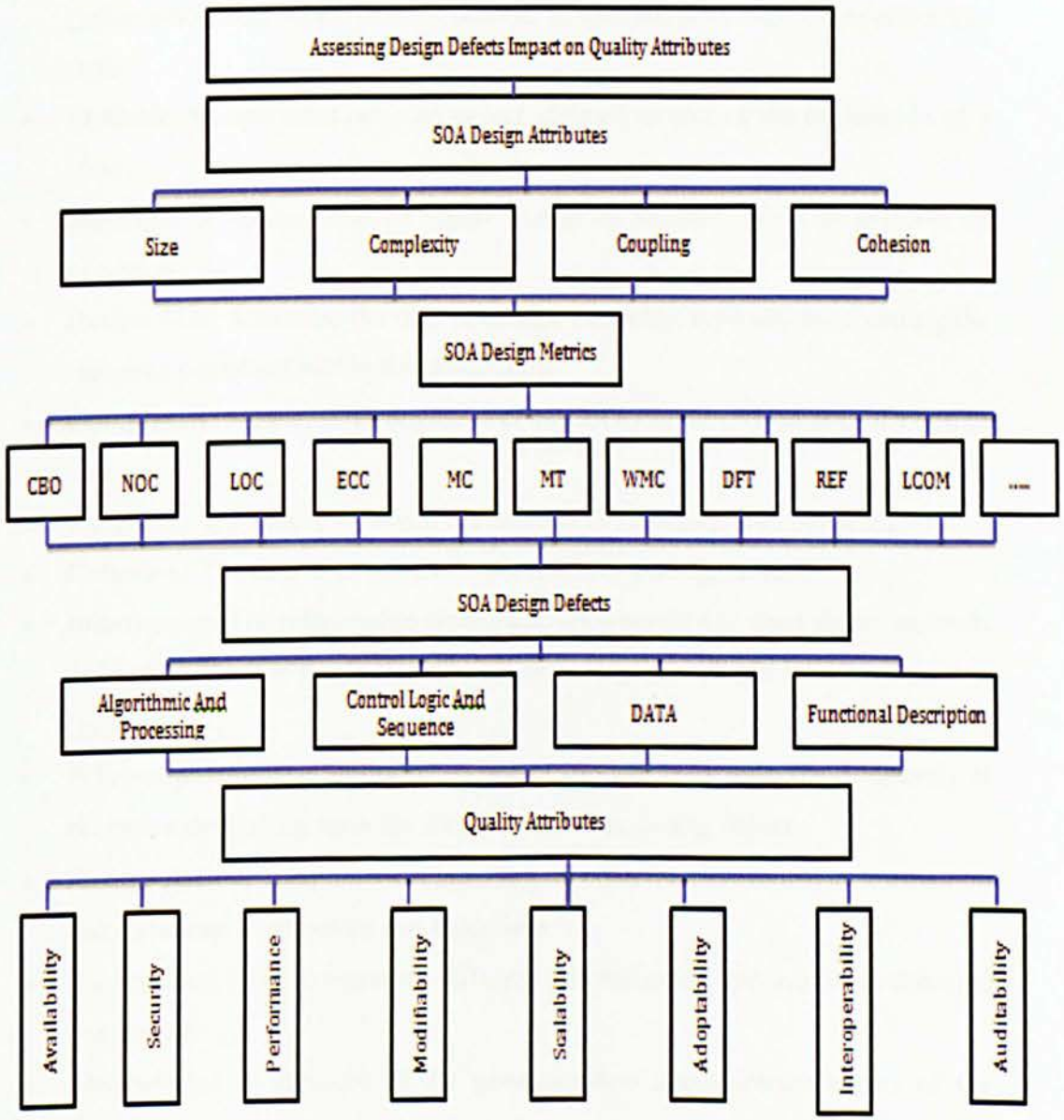

Figure 5.4: Relations Outline

5.3.3 Preparation Phase

The preparation phase consists of six main steps as follows.

The first step in preparing the "Design Defects Measuring Matrix" is to select the most common design attributes as shown in figure 5.5, these attributes are:

- **Class:** A set of objects that share a common structure and common behaviour manifested by a set of methods.
- **Object:** An instantiation of some class which is able to save a state (information) and which offers a number of operations to examine or affect this state.
- **Method:** An operation upon an object, defined as part of the declaration of a class.
- **Message:** A request that an object makes of another object to perform an operation.
- **Design Size:** Measures the size of design elements, typically by counting the elements contained within the design unit.
- **Complexity:** Measures the degree of connectivity between elements of a design unit.
- **Coupling:** The degree to which the elements in a design are connected.
- **Cohesion:** The degree to which the elements in a design unit.
- **Inheritance:** The relationship among classes wherein one class shares methods defined in one (for single inheritance) or more (for multiple inheritance) other classes.
- **Polymorphism:** The ability of an object to interpret a message differently at execution depending upon the super class of the calling object.
- **Encapsulation:** The process of bundling together the elements of an abstraction that constitute its structure and behaviour.
- **Hierarchies:** Used to represent different generalization-specialization concepts in a design.
- **Abstraction:** A measure of the generalization specialization aspect of the design.

- **Instantiation:** The process of creating an instance of the object and binding or adding the specific data.

SOA Design Attribute	Metrics	SOA Design Defects	SOA Quality Attributes
Size			
Complexity			
Coupling			
Cohesion			

Figure 5.5: The First Step in Preparing the Proposed Design Defects Measuring Matrix

The second step is to define the suitable metrics used to describe the selected design attributes as shown in figure 5.6. The following metrics are the most common metrics:

- Lines-Of-Code metric (LOC)
- Depth of Inheritance Tree (DIT)
- Source Line of Code (SLOC)
- Number Of Children (NOC)
- Halstead's Complexity (HC)
- Method Inheritance Factor (MIF)
- Attribute Inheritance Factor (AIF)
- Method Hiding Factor (MHF)
- Weighted Methods per Class (WMC)
- Response set For a Class (RFC)
- Coupling Between Object (CBO)
- Cyclomatic Complexity (CC)
- Maintainability Index (MI)
- Lack of Cohesion of Methods (LCOM)
- Polymorphism Factor (PF)
- Attribute Hiding Factor (AHF)

Once we have narrowed and assigned the list of metrics to consider for each design attribute, we can go to the next step.

SOA Design Attribute	Metrics	SOA Design Defects	SOA Quality Attributes
Size	LOC, CC, ECC, HC, MI, WMC, DIT, NOC, CBO, RFC & LCOM		
Complexity			
Coupling			
Cohesion			
...			
...			
...			
...			
...			
...			

Figure 5.6: The Second Step in Preparing the Proposed Design Defects Measuring Matrix

The third step is to define the most common design defects as shown in figure 5.7, these design defects are:

- **Algorithmic and Processing Defects that** occur when the processing steps in the algorithm as described by the pseudo code are incorrect.
- **Control, Logic and Sequence Defects that** occur when logic flow in the pseudo code is not correct. Logic defects usually related to incorrect use of logic operators.

- **Omission** it means that necessary information about the system has been omitted from the software artefact.
- **Incorrect Fact** it means that some information in the software artefact contradicts information in the requirements document or the general domain knowledge.
- **Inconsistency** it means that information within one part of the software artefact is inconsistent with other information in the software artefact.
- **Ambiguity** it means that information within the software artefact is ambiguous.
- **Extraneous Information** it means that information is provided that is not needed or used.
- **Functional Description Defects that** include incorrect, missing, and/or unclear design elements.
- **Intra-class Defects that** includes any design defect related to the internal structure of a class.
- **Behavioral Defects** it means all design defects related to the application semantics.
- **Inter-class Defects** it encloses any design defect related to the external structure of the classes and their relationships.

SOA Design Attribute	Metrics	SOA Design Defects	SOA Quality Attributes
Size	LOC CC, ECC, HC, MI, WMC, DIT, NOC, CBO, RFC & LCOM	Algorithmic and Processing	
Complexity		Control, Logic, and Sequence	
Coupling		Data	
Cohesion		Functional Description	

Figure 5.7: The Third Step in Preparing the Proposed Design Defects Measuring Matrix

Once we have narrowed and assigned the list of design defects, we can go to the next step.

The fourth step is to define the most common quality attributes as shown in figures 5.8. The quality attributes are:

- **Adaptability:** The quality of being adaptable to changes.
- **Analysability:** The capability of the software product to be diagnosed for deficiencies or causes of failures in the software.
- **Auditability:** The quality factor representing the degree to which an application or component keeps sufficiently adequate records to support one or more specified financial or legal audits.
- **Availability:** The ability of the user community to access the service, whether to submit a new request, update or alter existing request, or collect the results of a previous request.
- **Changeability:** The capability of the software product to enable a specified modification to be implemented.
- **Correctness:** The accountability for satisfying all requirements of the system. Measure of exact adherence to specifications.
- **Extensibility:** Extending an SOA by adding new services or incorporating additional capabilities into existing services is supported within an SOA.
- **Interoperability:** The ability to exchange and use information (usually in a large heterogeneous network made up of several local area networks).
- **Modifiability:** How the system can accommodate anticipated and unanticipated changes and is largely a measure of how changes can be made locally, with little ripple effect on the system at large.
- **Performance:** Refers to the system responsiveness: either the time required responding to specific events, or the number of events processed in a given time interval.
- **Reusability:** The degree to which a software module or other work product can be used in more than one computing program or software system.
- **Scalability:** The ability of SOA to function well when the system is changed in size or in volume in order to meet users' needs.

- **Stability:** The capability of the software product to avoid unexpected effects from modifications of the software.
- **Testability:** Can be negatively impacted when using an SOA due to the complexity of the testing services that are distributed across a network.
- **Understandability:** The degree to which the purpose of the system or component is clear to the evaluator.
- **Usability:** May decrease if the services within the application support human interactions with the system and there are performance problems with the services.

Once we have narrowed and assigned the most common quality attributes, we can go to assign and matching between all of them in the next steps.

SOA Design Attribute	Metrics	SOA Design Defects	SOA Quality Attributes							
			Availability	Security	Performance	Modifiability	Scalability	Adaptability	Interoperability	Auditability
Size	LOC CC, ECC, HC, MI, WMC, DIT, NOC, CBO, RFC & LCOM	Algorithmic and Processing								
Complexity		Control, Logic, and Sequence								
Coupling		Data								
Cohesion		Functional Description								

Figure 5.8: The Fourth Step in Preparing the Proposed Design Defects Measuring Matrix

The fifth step is to match between design attributes and design defects through the selected metrics and to match between design defects and quality attributes as shown in figure 5.9.

SOA Design Attribute	Metrics	SOA Design Defects	SOA Quality Attributes							
			Availability	Security	Performance	Modifiability	Scalability	Adaptability	Interoperability	Auditability
Size	LOC CC, ECC, HC, MI, WMC, DIT, NOC, CBO, RFC & LCOM	Algorithmic and Processing	■						■	
		Control, Logic, and Sequence				■				
		Data						■		
		Functional Description		■						
Complexity		Algorithmic and Processing			■		■			
		Control, Logic, and Sequence		■			■			
		Data	■							■

		Functional Description								
Coupling		Algorithmic and Processing								
		Control, Logic, and Sequence								
		Data								
		Functional Description								
Cohesion		Algorithmic and Processing								
		Control, Logic, and Sequence								
		Data								
		Functional Description								

Figure 5.9: The Fifth Step in Preparing the Proposed Design Defects Measuring Matrix

The sixth step is to assign the suitable metrics that can be used to measure the impact of design defects on quality attributes as shown in figure 5.10.

SOA Design Attribute	Metrics	SOA Design Defects	SOA Quality Attributes							
			Availability	Security	Performance	Modifiability	Scalability	Adaptability	Interoperability	Auditability
Size	LOC CC, ECC, HC, MI, WMC, DIT, NOC, CBO, RFC & LCOM	Algorithmic and Processing	CC						LOC	
		Control, Logic, and Sequence				ECC				
		Data						MI		
		Functional Description		HC						
Complexity		Algorithmic and Processing								
		Control, Logic, and Sequence								
		Data								

		Functional Description								
Coupling		Algorithmic and Processing								
		Control, Logic, and Sequence								
		Data								
		Functional Description								
Cohesion		Algorithmic and Processing								
		Control, Logic, and Sequence								
		Data								
		Functional Description								

Figure 5.10: The Sixth Step in Preparing the Proposed Design Defects Measuring Matrix

5.3.4 Application Phase

After preparing the "Design Defects Measuring Matrix", the following formula for measuring the impacts of SOA design defects on quality attributes can be used:

SOA design Defect Impact = Summation of Metrics ranks X Attributes Weights

SOA design Defect Impact =

$$\sum_{i=1}^{n} \sum_{j=1}^{mi} (MI_i \quad X \quad WA_{ij} \;)/mij \qquad \textbf{Eq. (5.1)}$$

Where:

MI_i	≡	Metric Rank
AW	≡	Quality Attribute Weight
i	≡	Metric number
j	≡	Attribute number measured by metric i
n	≡	Total number of Metrics in each SOA Design Defect
mi	≡	Total number of Attributes affected by metric i
mij	≡	Total number of Attributes affected by metrics

To calculate the weight of each quality attributes (AW); a scale of 100 can be used according to its impacts on SOA Quality as shown in figure 5.11.

SOA Quality Attributes	Availability	Security	Performance	Modifiability	Scalability	Adaptability	Interoperability	Auditability	Total
Weights (Out of 100)									100

Figure 5.11: Attributes Weights

Once we have designed the Defects Measuring Matrix, Attributes Weights and Metric Rank, we can use the DESQA framework to calculate SOA Design Defect Impacts.

5.4 Framework Design Execution

Having previously identified and formalized the different aspects of the framework in this section the application of the framework is described using number of tools and technologies as following the high level architecture and design and functionalities as presented in sections 5.3.1–5.3.2 and shown in figures 5.5–5.11.

The process starts with the preparation phase in which the requirements specification leading to a one or more design solutions to be considered. The design description is then created reflecting the actual design. Next, an intelligent parser is used on the design description in order to identify the potential defects and to create a defect portfolio "Design Defects Measuring Matrix".

Once this is done the values are collected in DB to produce the type of high level defects (pattern) and quality attributes and the framework is then ready to be applied for defect detection and quality estimation. In order to evaluate the framework, next to design description, number of technologies can be used such as Visual studio C#, UML, parsing etc.

5.4.1 Visual |Studio C# Advantages

The choice between C# and VB.NET is largely one of subjective preference. Some people like C#'s terse syntax, others like VB.NET's natural language, case-insensitive approach. Both have access to the same framework libraries. Both will perform largely equivalently. The following are the reasons for using Visual studio C#:

- XML documentation generated from source code comments.
- Operator overloading.
- Language support for unsigned types.
- The using statement, which makes unmanaged resource disposal simple.

- Explicit interface implementation, where an interface which is already implemented in a base class can be re-implemented separately in a derived class. Arguably, this makes the class harder to understand, in the same way that member hiding normally does.
- Unsafe code. This allows pointer arithmetic etc, and can improve performance in some situations. However, it is not to be used lightly, as a lot of the normal safety of C# is lost (as the name implies). Note that unsafe code is still managed code, i.e. it is compiled to IL, JITted, and run within the CLR.

5.4.2 Design Steps Using Visual Studio C#

- Creating the Project
- Creating a Control Library Project
- Referencing the Custom Control Project
- Defining a Custom Control and Its Custom Designer
- Creating an Instance of the Custom Control
- Setting Up the Project for Design-Time Debugging
- Implementing the Custom Control
- Creating a Child Control for the Custom Control
- Create the Marquee Border Child Control
- Creating a Custom Designer to Shadow and Filter Properties
- Handling Component Changes
- Adding Designer Verbs to the Custom Designer
- Creating a Custom UI Type Editor
- Testing the Custom Control in the Designer

5.4.3 Code Generation from UML Class Diagrams

UML is a general-purpose modeling language in the field of software engineering, which is designed to provide a standard way to visualize the design of a system. In order to evaluate the framework number of UML design diagrams can be used:

- **Class Diagram:** This shows a collection of static model elements such as classes, types, their contents, and their relationships.

- **Activity Diagram:** Which depicts high-level business processes, including data flow, and complex logic within a system.
- **Component Diagram:** This depicts the components/services that compose the application, their interrelationships, interactions, and their public interfaces.
- **Deployment Diagram**: This shows the execution architecture of systems.

However, the main focus in this work will be on Class and Component diagrams.
To generate Visual C# .NET code from UML class diagrams in Visual Studio Ultimate, the Generate Code command is used. By default, the command generates a C# type for each UML type that is selected. In addition it is possible to modify and extend this behaviour by modifying or copying the text templates that generate the code. Moreover, it is possible to specify different behaviour for the types that are contained in different packages in the model.

The Generate Code command is particularly suited to generating code from the user's selection of elements, and to generating one file for each UML class or other element. For example, the screenshot shows two C# files that have been generated from two UML classes.

As an alternative, it is possible to generate code in which the generated files do not have a 1:1 relationship with the UML elements, and writing text templates that are invoked with the Transform All Templates command can be considered [116].

5.4.4 Parsing Work

The task of the parser is essentially to determine if and how the input can be derived from the start symbol of the grammar. This can be done in essentially two ways:

Top-down parsing: Top-down parsing can be viewed as an attempt to find left-most derivations of an input-stream by searching for parse trees using a top-down expansion of the given formal grammar rules. Tokens are consumed from left to right. Inclusive choice is used to accommodate ambiguity by expanding all alternative right-hand-sides of grammar rules [117].

Some of the parsers that use top-down parsing include:

- Recursive descent parser
- LL parser (**L**eft-to-right, **L**eftmost derivation)
- Earley parser.

Bottom-up parsing: A parser can start with the input and attempt to rewrite it to the start symbol. Intuitively, the parser attempts to locate the most basic elements, then the elements containing these, and so on. LR parsers are examples of bottom-up parsers. Another term used for this type of parser is Shift-Reduce parsing.

Some of the parsers that use bottom-up parsing include:

- Precedence parser
  - Operator-precedence parser
  - Simple precedence parser
- BC (bounded context) parsing
- LR parser (Left-to-right, Rightmost derivation)
- Simple LR (SLR) parser
- LALR parser
- Canonical LR (LR(1)) parser
- GLR parser
- CYK parser
- Recursive ascent parser
- Shift-Reduce parser

The first stage is the token generation as shown in figure 5.12, by which the input character stream is split into meaningful symbols defined by a grammar of regular expressions.

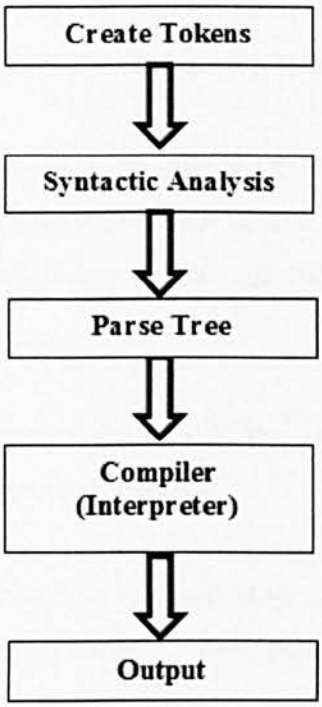

Figure 5.12: Parsing Process [118]

The next stage is parsing or syntactic analysis, which is checking that the tokens form an allowable expression. This is usually done with reference to a context-free grammar which recursively defines components that can make up an expression and the order in which they must appear. The final phase is semantic parsing or analysis, which is working out the implications of the expression just validated and taking the appropriate action. C# has some of the best text libraries out there. Parser is created based on the identified metrics like:

-LOC	- CC	- ECC	- HC	- MI
-WMC	- DIT	- NOC	- CBO	-RFC
LCOM				

5.5 Summary

Developing quality code is a major concern for the software community. Defect prevention is an important activity in any software quality. The main objective of this work is to propose a new framework to measure software quality. The DESQA framework is a comprehensive, multidimensional framework of SOA defects detection.

It consists of seven major components: requirements, design, description, intelligent parser, defect database, defects portfolio and quality assessment.

The DESQA framework can be used to calculate SOA design Defect Impacts in Design phase only (Conceptual, Preliminary and Final design phases), and all SOA design attributes, quality metrics, SOA design defects and SOA quality attributes are well defined and can be measured.

To formalize and build the proposed framework, the following measures are defined and adapted primarily from previous researches:

- Eleven items were selected to measure SOA Design Defects (for example: Algorithmic and Processing Defects, Omission, Incorrect Fact, Inconsistency, Functional Description Defects ...)
- Fourteen items were selected to measure SOA Design Attributes (for example: Design Size, Complexity, Coupling, Cohesion...)
- Seventeen items were selected to measure SOA Quality Attributes (for example: Correctness, Modifiability, Performance, Usability, Reusability, Scalability, Stability, Testability, Understandability ...)
- Eleven items were selected to measure SOA Quality Metrics (for example: LOC, WMC, DIT, RFC, CBO, NOC, CC, LCOM...).

The process of application starts with defining Attributes weights and Metric Rank before using the Defects Measuring Matrix. In the preparation phase the requirements specification leading to one or more design solutions needs to be considered. Once this is done the values are collected in DB to produce the type of high level defects (pattern) and quality attributes and the framework is then ready to be applied in the application phase. After designing Defects Measuring Matrix, Attributes Weights and Metric Rank, we can use the proposed formula to calculate SOA Design Defect Impacts.

CHAPTER 6: CASE STUDY AND EVALUATION

6.1 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 chapter 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.

In this sense, this chapter starts with a preliminary evaluation performed with the purpose of evaluating the usability, understanding and applicability of the proposed framework. The goal of this work is to investigate the relationship between several software metrics, the design defects and software quality. This chapter discusses the building process and the use of Design Defects Measuring Matrix as a mean of helping assessing software quality. Most of the metrics discussed in this chapter are not difficult to compute. In addition, the evaluation using a case study aims to demonstrate the use of the framework on a number of designs and produces an overall picture regarding defects and quality.

The rest of the chapter is organized as follows. Section 6.2 describes the research tool (questionnaire), the sample used to complete the Design Defects Measuring Matrix, and data collection process. Section 6.3 describes the results of the analysis process. Section 6.4 presents a case study together with the main findings. Finally, a conclusion is presented in section 6.5.

6.2 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. The questionnaire consists of two parts as follows (See appendix A):

Part (I): Definitions

- Definition of Design Defects
- Definition of Design Attributes
- Definition of Metrics
- Definition of Quality Attribute

Part (II): Selection Process

- Identification of Relation between Design Attributes, SOA Design Defects, Quality Metrics and Quality Attributes

Part (III): Metrics Measurement Range

- Mapping the relationship between SOA Quality Attribute and Quality Metrics

6.2.1 Sample Used

To demonstrate the usability of the proposed "Design Defects Measuring Matrix", we have designed and implemented two different scenarios (building and using the matrix). For the purpose of this study, two conditions were applied to select the participant companies: experience and acceptance to participate. Five software companies working in Kuwait were selected based on their experiences. After personal contact, three companies agreed to participate in the study with the condition that we hide their names. Each company nominated four experts. These experts divided into two groups; the first

group consists of nine software designers working in software development. The second group consists of three programmers. All participants were trained in a one-day workshop on how to use the "Design Defects Measuring Matrix", after that the first group used the proposed questionnaire to construct the matrix and the second one used the final matrix.

6.2.2 Data Collection

After having presented and introduced general definitions of SOA, as well as some detailed information about different attributes of the architecture, the first group was asked to select the most used items. The results of the selection process are as follows and as summarized in table (6-1):

- **The design attributes**
  Size, complexity, coupling, and cohesion.
- **The SOA Design Defects**
  Algorithmic and Processing Defects, Control, Logic, and Sequence Defects, Omission, Incorrect Fact, Inconsistency and Functional Description Defects.
- **The metrics**
  Since specific metrics for measuring software attributes of SOA-based systems are yet to be defined, one of the objectives of this work was to assess the applicability of conventional software engineering metrics to SOA. A set of seven well-established metrics was chosen based on their importance and applicability to SOA approaches. For the purpose of this study the metrics usage are as follows:
  1) For Size: Lines of Code (LOC) that constitute the system.
  2) For Complexity:
     a) Traditional Cyclomatic Complexity (CC)
     b) Weighted Method per Class (WMC)
     c) Depth of Inheritance Tree (DIT) and Number of Children (NOC)
  3) For Coupling:
     a) Coupling between Objects (CBO)
     b) Response for Class (RFC)
  4) For Cohesion: lack of cohesion of methods (LCOM).

- **The quality attributes**

  The first step is the identification of attributes of the qualities of SOA; the group looks at each quality attributes listed in Table 3.1. The results show that:

  - The following attributes are favoured by **simple solutions** — Testability, Flexibility, Portability, Changeability, Reusability, Stability and Analysability
  - The following attributes are favoured by **general solutions** — Flexibility and Reusability
  - The following attributes are favoured by having a **modular design** — Testability, Flexibility, Reusability and Analysability
  - The following attributes are favoured by **designing with change in mind** — Flexibility, Changeability, Stability and Analysability
  - The following attributes are favoured by using a **middleware system** — Interoperability, Reusability and Testability
  - The following attributes are favoured by having **traceability** between system artefacts: Correctness and Analysability
  - The following attributes are favoured by **low coupling** between components or modules: Changeability, Stability and Testability

We see that using *simple solutions*, having a *modular design* and *designing for change* are three approaches that facilitate most of the quality attributes. The group suggested that the following attributes are the most influential quality attributes (Final Quality Attributes):

Correctness: which appears in McCall's quality model, can be seen as a developer-oriented quality attribute given that it should be relevant for developers that seek to ease their efforts in developing and maintaining the system. McCall's model links Correctness to three quality criteria, i.e. characteristics: Traceability, Completeness and Consistency.

Modifiability: which appears in Boehm's model, the degree to which a system or component facilitates the incorporation of changes, once the nature of the desired change has been determined.

Performance: performance can have different meanings in different contexts. In general, it is related to response time (how long it takes to process a request), throughput (how many requests overall can be processed per unit of time), or timeliness (ability to meet deadlines, i.e., to process a request in a deterministic and acceptable amount of time). Performance is an important quality attribute that is usually affected negatively in SOAs. An SOA approach can have a negative impact on the performance of an application due to network delays, the overhead of looking up services in a directory, and the overhead caused by intermediaries that handle communication. The service user must design and evaluate the architecture carefully, and the service provider must design and evaluate its services carefully to make sure that the necessary performance requirements are met.

Usability, which appears in ISO/IEC 9126 quality model, Usability may decrease if the services within the application support human interactions with the system and there are performance problems with the services. It is up to the service users and providers to build support for usability into their systems.

Reusability, which appears in McCall's model, and is decomposed into the following characteristics: Simplicity, Generality, Modularity, Software system independence and Machine independence.

Scalability, which is the ability of an SOA to function well (without degradation of other quality attributes) when the system is changed in size or in volume in order to meet users' needs. There are ways to deal with an increase in the number of service users and the increased need to support more requests for services. However, these solutions require detailed analysis by the service providers to make sure that other quality attributes are not negatively impacted. Options for solving scalability problems include

- Horizontal scalability: distributing the workload across more computers. Doing so may mean adding an additional tier or more service sites.
- Vertical scalability: upgrading to more powerful hardware for the service site.

Stability, which appears in ISO/IEC 9126 quality model, Stability is in this context not directly connected to the ability of the system to show stable behaviour when used. However, if modifications often have unexpected effects, then system's Stability from a use perspective will be affected. Stability is related to Changeability, in that a system with low Changeability is likely to show low Stability as well. This follows from the fact that trying to modify a system with low Changeability is associated with great risk and can result in faults.

Testability, which appears in ISO/IEC 9126 quality model, Testability is an attribute that occurs in both McCall's model and Boehm's model. In the models, it corresponds to the following characteristics: Simplicity, Instrumentation, Self-descriptiveness, Modularity and structuredness, Accountability, Accessibility and Communicativeness.

Understandability, which appears in Boehm's model, Understandability is the degree to which a system or component facilitates the incorporation of changes, once the nature of the desired change has been determined.

Based on the discussions below, table (6.1) summarizes the selection results. After selecting the items (SOA Design Attribute, Metrics, SOA Design Defects and SOA Quality Attributes), the group was asked to give weights to the selected SOA Quality Attributes. The group gave weights out of 100 as shown in table (6.2).

Table (6.1): Selection Process Results

SOA Design Attribute	Metrics	SOA Design Defects	SOA Quality Attributes
Design Size Complexity Coupling Cohesion	LOC WMC DIT RFC CBO NOC CC LCOM	• Algorithmic and Processing Defects • Control, Logic, and Sequence Defects • Omission • Incorrect Fact • Inconsistency • Functional Description Defects	Correctness Modifiability Performance Usability Reusability Scalability Stability Testability Understandability

Table (6.2): SOA Quality Attributes Weights

SOA Quality Attributes	Correctness	Modifiability	Performance	Usability	Reusability	Scalability	Stability	Testability	Understandability	Total
Weights	**10**	**10**	**10**	**15**	**10**	**7**	**8**	**15**	**15**	**100**

Once the weights of Quality Attributes were defined, the group was asked to match between the quality metrics and quality attributes and give weights to the metrics impacts on quality out of 10 as shown in table (6.3).

Table (6.3): Metrics Measurement Range

Impact on Quality [(10) Highest impact (1) Lowest impact]

Quality Metrics	SOA Quality Attributes								
	Correctness	Modifiability	Performance	Usability	Reusability	Scalability	Stability	Testability	Understandability
LOC	6	5	4			5			4
WMC				7	4			3	
DIT					1			3	3
RFC				3				4	
CBO	1				2				
NOC	3		6			5			3
CC		5					2		
LCOM					3		8		
	10	10	10	10	10	10	10	10	10

The overall structure of the Design Defects Measuring Matrix is shown in Table (6.4).

Table (6.4): Design Defects Measuring Matrix

SOA Design Defects	SOA Quality Attributes								
	Correctness	Modifiability	Performance	Usability	Reusability	Scalability	Stability	Testability	Understandability
	10	10	10	15	10	7	8	15	15
Algorithmic and Processing Defects	LOC (6) CBO (1) NOC (3)		LOC (4) NOC (6)	WMC (7) RFC (3)	WMC (4) DIT (1) CBO (2) LCOM (3)	LOC (5) NOC (5)		WMC (3) DIT (3) RFC (4)	LOC (4) NOC (3) DIT (3)

Control, Logic, and Sequence Defects	**LOC (6) CBO (1) NOC (3)**		**LOC (4) NOC (6)**			**LOC (5) NOC (5)**	**CC (2) LCOM (8)**	**WMC (3) DIT (3) RFC (4)**	
Omission	**LOC (6) CBO (1) NOC (3)**	**LOC (5) CC (5)**		**WMC (7) RFC (3)**				**WMC (3) DIT (3) RFC (4)**	**LOC (4) NOC (3) DIT (3)**
Incorrect Fact	**LOC (6) CBO (1) NOC (3)**	**LOC (5) CC (5)**	**LOC (4) NOC (6)**	**WMC (7) RFC (3)**				**WMC (3) DIT (3) RFC (4)**	**LOC (4) NOC (3) DIT (3)**

Inconsistency	**LOC (6) CBO (1) NOC (3)**				**WMC (4) DIT (1) CBO (2) LCOM (3)**		**CC (2) LCOM (8)**	**WMC (3) DIT (3) RFC (4)**	
Functional Description Defects	**LOC (6) CBO (1) NOC (3)**	**LOC (5) CC (5)**			**WMC (4) DIT (1) CBO (2) LCOM (3)**	**LOC (5) NOC (5)**		**WMC (3) DIT (3) RFC (4)**	**LOC (4) NOC (3) DIT (3)**

6.3 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 then used the designed Design Defects Measuring Matrix shown in table (6.4) to assess the impacts of expected design defects on quality. 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: Correctness, Performance, Usability, Reusability, Scalability, Testability and Understandability as shown in table (6.5).

The quality metrics used to assess the impacts on quality are shown in table (6.6). The weights of metrics and quality attributes are used to calculate the total impact on quality.

Table (6.5): Impacts of Algorithmic and Processing Defects and Quality Attributes

SOA Design Defects	SOA Quality Attributes						
	Correctness	Performance	Usability	Reusability	Scalability	Testability	Understandability
	10	10	15	10	7	15	15
Algorithmic and Processing Defects	LOC (6) CBO (1) NOC (3)	LOC (4) NOC (6)	WMC (7) RFC (3)	WMC (4) DIT (1) CBO (2) LCOM (3)	LOC (5) NOC (5)	WMC (3) DIT (3) RFC (4)	LOC (4) NOC (3) DIT (3)

Table (6.6): Quality Metrics Used to Assess the Impacts on Quality

Quality Metrics	Impact on Quality
LOC	Correctness + Performance + Scalability (6*10 + 4*10 + 5*7 + 4*15) / 4 = 195/4
WMC	Usability + Reusability + Testability (7*15 + 4*10 + 3*15) / 3 = 140/3
DIT	Reusability + Testability + Understandability (1*10 + 3*15 + 3*15) / 3 = 100/3
RFC	Usability + Testability (3*15 + 4*15) / 2 = 105/2
CBO	Correctness + Reusability (1*10 + 2*10) / 2 = 30/2
NOC	Correctness + Performance + Scalability + Understandability (3*10 + 6*10 + 5*7 + 3*15) / 4 = 170/4
CC	0
LCOM	Reusability (3*10) / 1 =30
Total Impact on Quality = % 38.39	

It means that Algorithmic and Processing Defects will reduce the quality by % 38.39 and the expected quality of the design = 100 – 38.39 = % 61.61.

Similarly Defects related to Control, Logic, and Sequence Defects will affect the following quality attributes: Correctness, Performance, Scalability, Stability and Testability as shown in table (6.7). Quality metrics used to assess the impacts on quality are shown in table (6.8).

Table (6.7): Impacts of Control, Logic, and Sequence Defects and Quality Attributes

SOA Design Defects	SOA Quality Attributes				
	Correctness	Performance	Scalability	Stability	Testability
	10	10	7	8	15
Control, Logic, and Sequence Defects	LOC (6) CBO (1) NOC (3)	LOC (4) NOC (6)	LOC (5) NOC (5)	CC (2) LCOM (8)	WMC (3) DIT (3) RFC (4)

Table (6.8): Quality Metrics Used to Assess the Impacts on Quality

Quality Metrics	Impact on Quality
LOC	(6*10 + 4*10 + 5*7) / 3 = 135/3
WMC	(3*15) / 1 = 45
DIT	(3*15) / 1 = 45
RFC	(4*15) / 1 = 60
CBO	(1*10) / 1 = 10
NOC	(3*10 + 6*10 + 5*7) / 3 =125/3
CC	(2*8) / 1 = 16
LCOM	(8*8) / 1 = 64
Total Impact on Quality = % 43.33	

It means that Control, Logic, and Sequence Defects will reduce the quality by % 43.33 and the expected quality of the design = 100 – 43.33 = % 56.67.

Similarly Defects related to Omission Defects will affect the following quality attributes: Correctness, Modifiability, Usability, Testability and Understandability as shown in table (6.9). Quality metrics used to assess the impacts on quality are shown in table (6.10).

Table (6.9): Impacts of Omission Defects and Quality Attributes

SOA Design Defects	SOA Quality Attributes				
	Correctness	Modifiability	Usability	Testability	Understandability
	10	10	15	15	15
Omission	LOC (6) CBO (1) NOC (3)	LOC (5) CC (5)	WMC (7) RFC (3)	WMC (3) DIT (3) RFC (4)	LOC (4) NOC (3) DIT (3)

Table (6.10): Quality Metrics Used to Assess the Impacts on Quality

Quality Metrics	Impact on Quality
LOC	(6*10 + 5*10 + 4*15) / 3 = 170/3
WMC	(7*15 + 3*15) / 2 = 150/2
DIT	(3*15 + 3*15) / 2 = 90/2
RFC	(3*15 + 4*15) / 2 =105/2
CBO	(1*10) / 1 = 10
NOC	(3*10 + 3*15) / 2 = 75/2
CC	(5*10) / 1 = 50
LCOM	0
Total Impact on Quality = % 47.67	

It means that Omission Defects will reduce the quality by % 47.67 and the expected quality of the design = 100 – 47.67 = % 53.33

Similarly Defects related to Incorrect Fact Defects will affect the following quality attributes: Correctness, Modifiability, Performance, Usability, Testability and Understandability as shown in table (6.11). Quality metrics used to assess the impacts on quality are shown in table (6.12).

It means that Incorrect Fact Defects will reduce the quality by % 45.00 and the expected quality of the design = 100 – 45.00 = % 55.00

Table (6.11): Impacts of Incorrect Fact Defects and Quality Attributes

SOA Design Defects	SOA Quality Attributes					
	Correctness	Modifiability	Performance	Usability	Testability	Understandability
	10	10	10	15	15	15
Incorrect Fact	LOC (6) CBO (1) NOC (3)	LOC (5) CC (5)	LOC (4) NOC (6)	WMC (7) RFC (3)	WMC (3) DIT (3) RFC (4)	LOC (4) NOC (3) DIT (3)

Table (6.12): Quality Metrics Used to Assess the Impacts on Quality

Quality Metrics	Impact on Quality
LOC	(6*10 + 5*10 + 4*10 + 4*15) / 4 = 210/4
WMC	(7*15 + 3*15) / 2 = 150/2
DIT	(1*10 + 3*15 + 3*15) / 3 = 100/3
RFC	(3*15 + 4*15) / 2 = 105/2
CBO	(1*10) / 1 = 10
NOC	(3*10 + 6*10 + 3*15) / 3 =125/3
CC	(5*10) / 1 = 50
LCOM	0
Total Impact on Quality = % 45.00	

Similarly Defects related to Inconsistency Defects will affect the following quality attributes: Correctness, Reusability, Stability and Testability Standability as shown in table (6.13). Quality metrics used to assess the impacts on quality are shown in table (6.14).

Table (6.13): Impacts of Inconsistency Defects and Quality Attributes

SOA Design Defects	SOA Quality Attributes			
	Correctness	Reusability	Stability	Testability
	10	10	8	15
Inconsistency	LOC (6) CBO (1) NOC (3)	WMC (4) DIT (1) CBO (2) LCOM (3)	CC (2) LCOM (8)	WMC (3) DIT (3) RFC (4)

Table (6.14): Quality Metrics Used to Assess the Impacts on Quality

Quality Metrics	Impact on Quality
LOC	(6*10) / 1 = 60
WMC	(4*10 + 3*15) / 2 = 85/2
DIT	(1*10 + 3*15) / 2 = 55/2
RFC	(4*15) / 1 = 60
CBO	(1*10 + 2*10) / 2 = 30/2
NOC	(3*10) / 1 = 30
CC	(2*8) / 1 = 16
LCOM	(3*10) / 1 =30
Total Impact on Quality = % 38.39	

It means that Inconsistency Defects will reduce the quality by % 35.13 and the expected quality of the design = 100 – 35.13 = % 64.87

Similarly Defects related to Functional Description Defects will affect the following quality attributes: Correctness, Modifiability, Reusability, Scalability, Testability and Understandability as shown in table (6.15). Quality metrics used to assess the impacts on quality are shown in table (6.16).

Table (6.15): Impacts of Functional Description Defects and Quality Attributes

SOA Design Defects	SOA Quality Attributes					
	Correctness	Modifiability	Reusability	Scalability	Testability	Understandability
	10	10	10	7	15	15
Functional Description Defects	LOC (6) CBO (1) NOC (3)	LOC (5) CC (5)	WMC (4) DIT (1) CBO (2) LCOM (3)	LOC (5) NOC (5)	WMC (3) DIT (3) RFC (4)	LOC (4) NOC (3) DIT (3)

Table (6.16): Quality Metrics Used to Assess the Impacts on Quality

Quality Metrics	Impact on Quality
LOC	(6*10 + 5*10 + 5*7 + 4*15) / 4 = 205/4
WMC	(4*10 + 3*15) / 2 = 85/2
DIT	(1*10 + 3*15 + 3*15) / 3 = 100/3
RFC	(4*15) / 1 = 60
CBO	(1*10 + 2*10) / 2 = 30/2
NOC	(3*10 + 5*7 + 3*15) / 3 =110/3
CC	(5*10) / 1 = 50
LCOM	(3*10) / 1 = 30
Total Impact on Quality = % 38.39	

It means that Functional Description Defects will reduce the quality by % 39.84 and the expected quality of the design = 100 – 39.84 = % 60.16.

6.4 Case Study: Automated Teller Machine (ATM)

This section demonstrates the applicability and use of the framework proposed in the previous chapter, 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) [119]. 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 [120]. 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 [121].

6.4.1 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.

6.4.2 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 6.1).

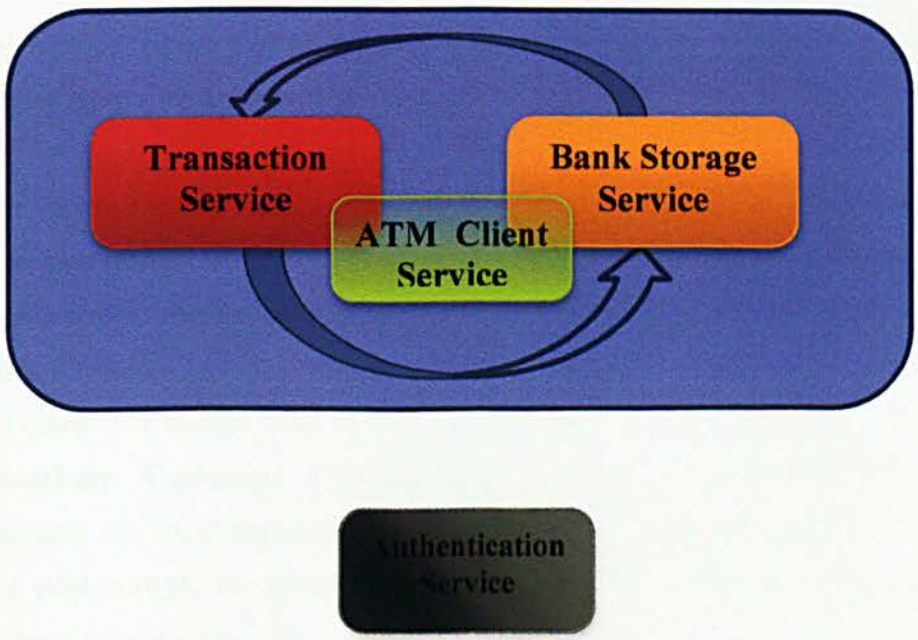

Figure 6.1: ATM System

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 6.2).

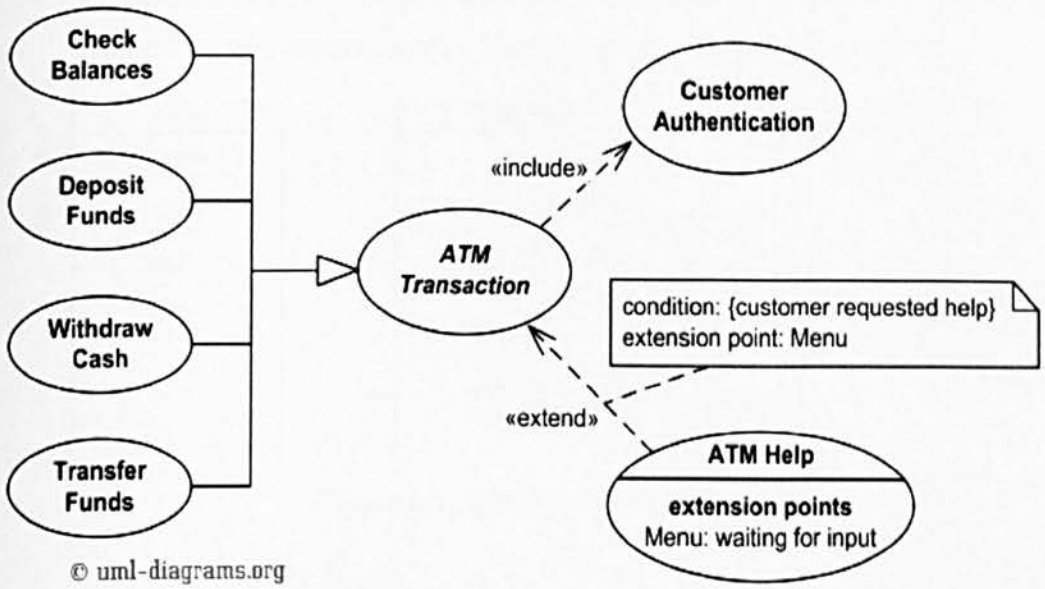

Figure 6.2 Use Case Diagram [122]

A transaction service (figure 6.3) shows a description of using an ATM machine to withdraw money from a bank account as follows:

- **Insert Card**: The use case begins when the customer inserts their bank card into the card reader of the ATM. The system allocates an ATM session identifier to enable errors to be tracked and synchronized between the ATM and the Bank System.
- **Read Card:** The system reads the bank card information from the card.
- **Authenticate Customer:** Perform Subflow Authenticate customer to authenticate the use of the bank card by the individual using the machine.
- **Select Withdrawal:** The system displays the service options that are currently available on the machine. The customer selects to withdraw cash.
- **Select Amount:** The system prompts for the amount to be withdrawn by displaying the list of standard withdrawal amounts. The customer selects an amount to be withdrawn.
- **Confirm Withdrawal:** Perform Subflow Assess Funds on Hand. Perform Subflow Conduct Withdrawal.

- **Eject Card:** The system ejects the customer's bank card. The customer takes the bank card from the machine.
- **Dispense Cash**: The system dispenses the requested amount of cash to the customer. The system records a transaction log entry for the withdrawal.

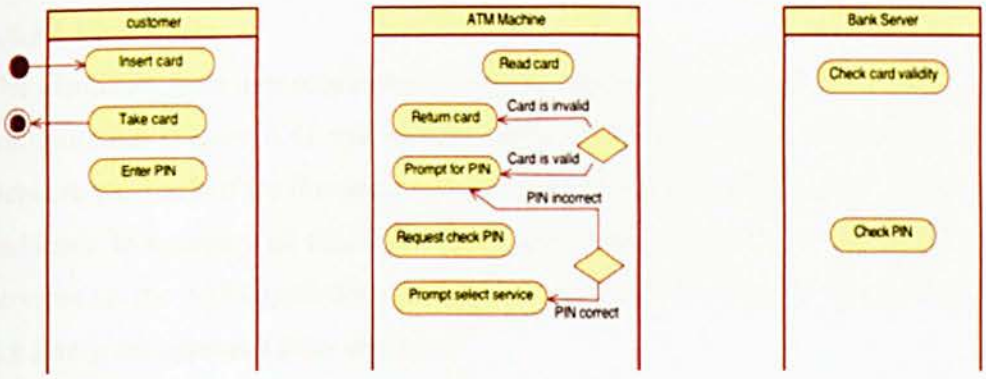

Figure 6.3: Transaction Service

6.4.3 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.

The types of granularity, if not handled properly, might give rise to different types of anti-patterns (please see chapter 4), mainly in the form of tiny service and multi service. These are two common anti-patterns in service-oriented systems, similar to the well-known anti-patterns in object-oriented systems. 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.

6.4.3.1 Fine Grain

The identified, from user requirements, service candidates are mapped to a typical SOA configuration (Figure 6.4) 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).

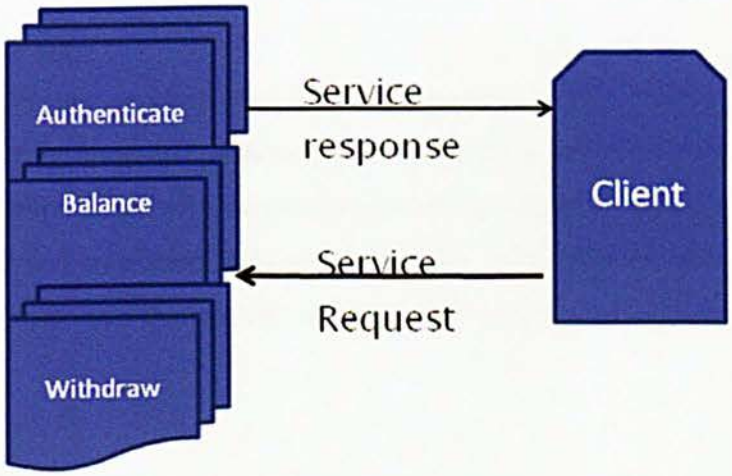

Figure 6.4 Fine Grain Services

6.4.3.2 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 6.5 the operations are represented by various components/services that follow the principles of SOA.

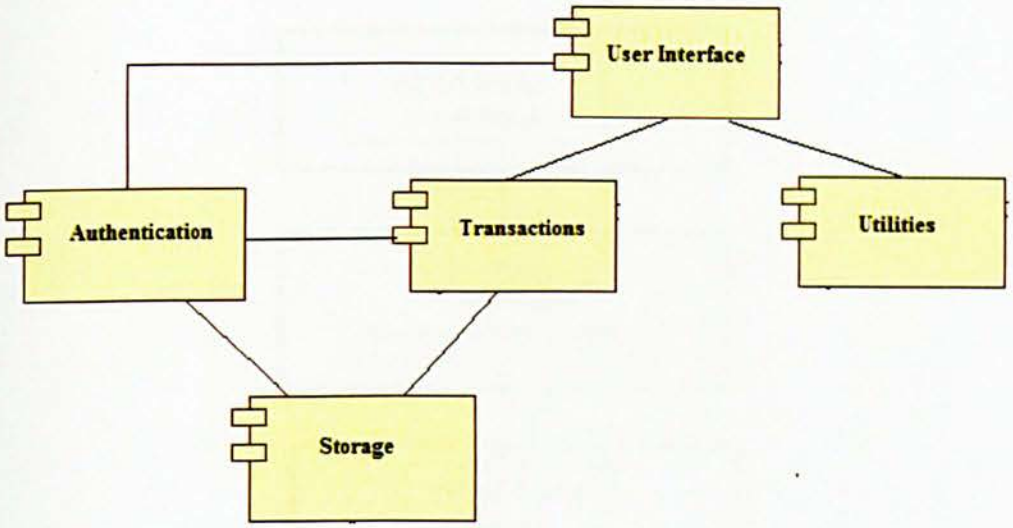

Figure 6.5 Coarse Grain Services

6.4.3.3 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 6.6.

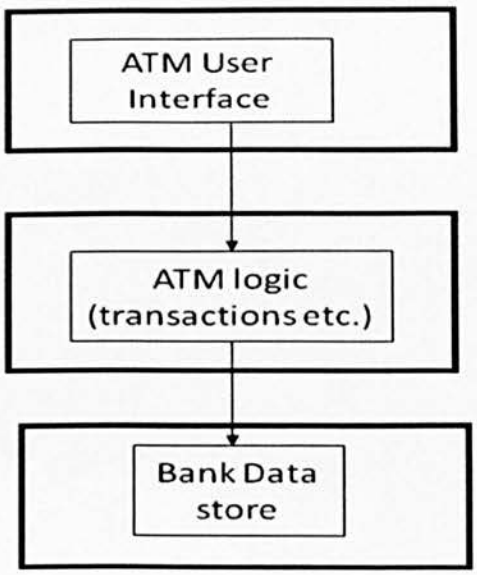

Figure 6.6 Three Tier Architecture

6.4.4 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 6.7, 6.8 and 6.9. In all experiments the following metrics have been used:

- The LOC was used to compare the size
- The CC, ECC, MI, WMC are used to compare the degree of complexity
- The CBO and RFC are used to compare the level of coupling
- The LCOM is used to compare the level of cohesion.

The metrics have been calculated using the equations presented in the LIST OF EQUATIONS (please refer to page 13 of this thesis).

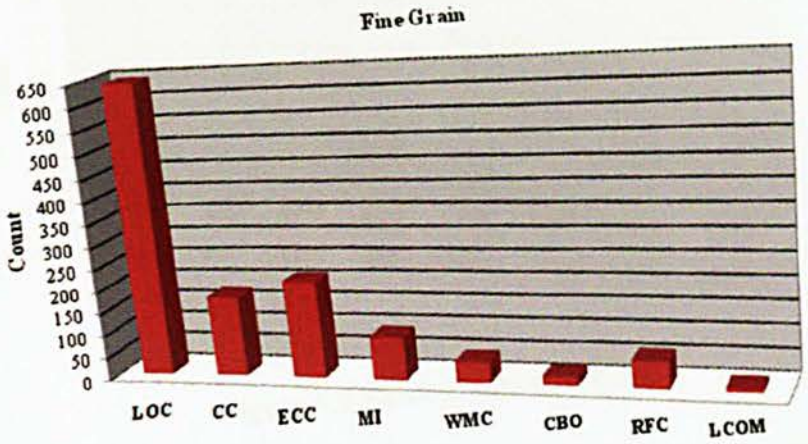

Figure 6.7 Fine Grain Services

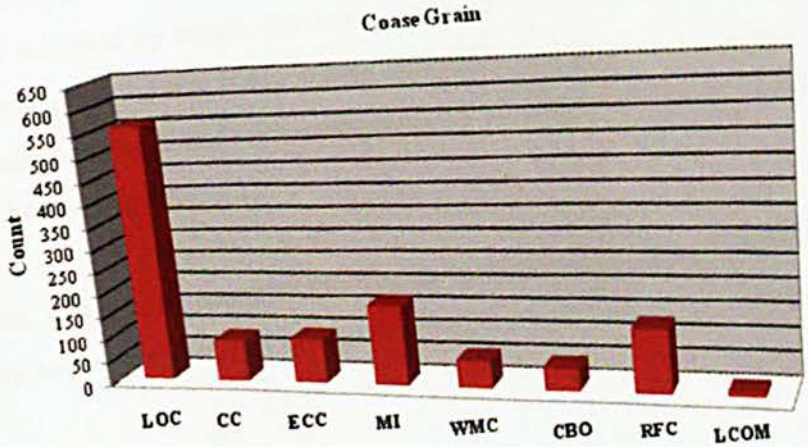

Figure 6.8 Coarse Grain Services

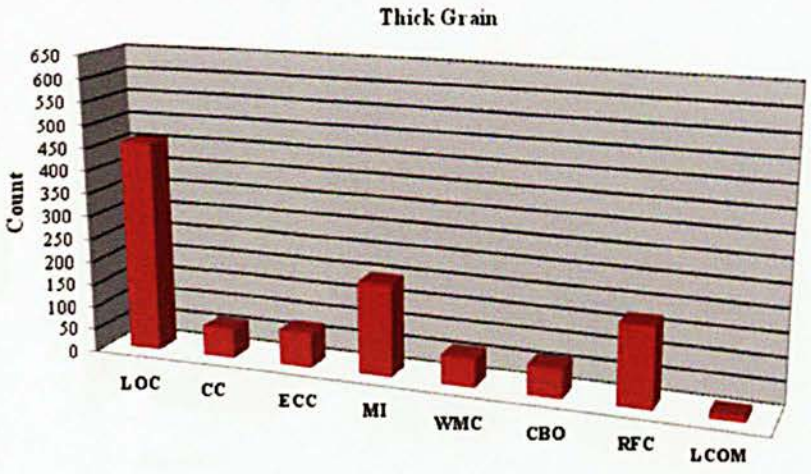

Figure 6.9 Thick Grain Services

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 6.10).

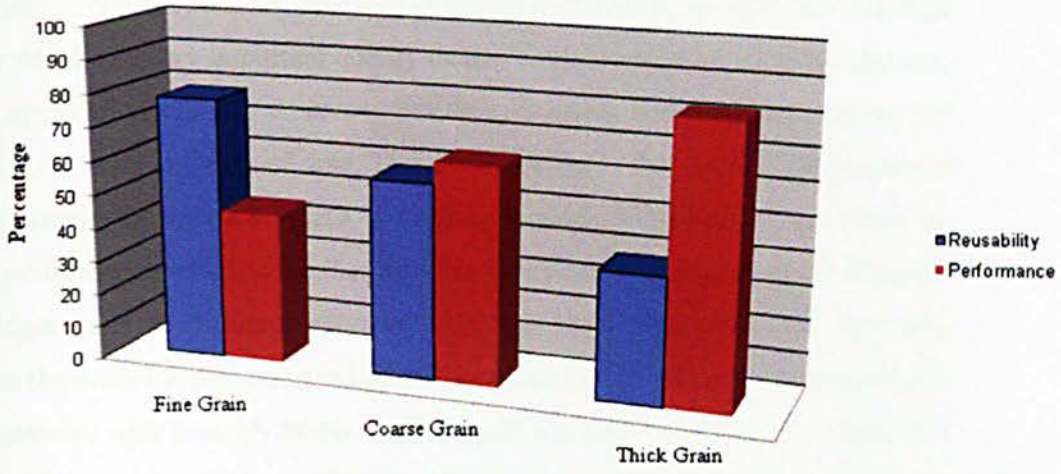

Figure 6.10 Software Quality Factors

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 6.11), but this will at the end depend very much on the type of applications and the user requirements.

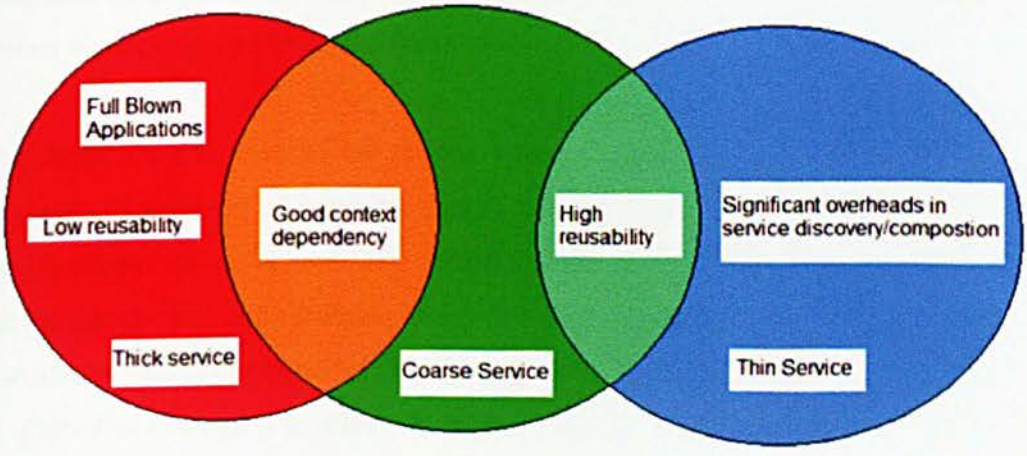

Figure 6.11 Different Granularity Impacts

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.

6.5 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 chapter 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.

CHAPTER 7: CONCLUSIONS, CRITICAL DISCUSSIONS AND FUTURE WORK

7.1 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.

7.2 Critical Discussions

The classification of defects found during software development plays an important role in measurement-based process and product improvement. The research also goes through the definition of defects in system development life cycle including *defects classification,* categories and detailed description of design defects. The design attributes are also discussed (class, object, method, message instantiation, inheritance, polymorphism, encapsulation, cohesion, coupling, design size, hierarchies, abstraction and complexity).

However, as pointed out earlier most of the reviewed techniques were developed for the detection of object-oriented design defects. Although, a number of SOA design defects were identified no attempts so far have been made to automate the process of detections of such defects and estimate quality. The main objective and contribution of this thesis is to address the defects in the software development life cycle process particularly at the design stages. To achieve this objective, a comprehensive and multidimensional framework of SOA defects detection was proposed. This framework examines the relationship between service-oriented architectures (SOAs) and quality attributes and outlines a set of quality attributes that may be derived from an organization's business goals and examines how those attributes relate to an SOA quality.

In addition, it describes how the SOA impacts those attributes and how choosing an SOA can help an organization estimate software quality based on its business goals. Finally, the framework presents guidelines for the automation process which is based on the intelligent application for the detecting SOA defects and estimating software quality factors. In this section the achievements and conclusions which have been previously drawn will be summarized.

The framework we proposed to study the design defects has a lot of potential. The correlation of metrics values with the number of quality attributes is an important step forward in the assessment of the quality of software design. Still, there is room for improvement. We have selected several defects metrics and software quality factors but we need to perform the same study on different scenarios. To evaluate the proposed framework, two groups representing some experts in the State of Kuwait were used to demonstrate the impact of quality metrics and quality attributes development on the design attributes of size, complexity, coupling, and cohesion.

The implementation part demonstrated how the framework can predict the quality level of the designed software. The results showed how 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 a more usable and maintainable software system to fulfil the demand of an efficient system for software applications. The overall quality value is then calculated by using the formula proposed in chapter 5.

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.

In addition, a real case study "Automated Teller Machine" was used to examine the validity of the proposed framework; we apply the framework using different levels of granularity, fine, coarse and thick. 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 and tends to promote higher reusability than larger grain styles. 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).

7.3 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 include large 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. Josuttis, N.M. (2007). SOA in Practice: The Art of Distributed System Design (Theory in Practice). O'Reilly.
11. Sanders, D.T., Hamilton Jr, J.A., & MacDonald, R.A. (2008, April). Supporting a service-oriented architecture. In *Proceedings of the 2008 Spring Simulation Multi-conference*, pp. 325-334. Society for Computer Simulation International.
12. Bertoa, M.F., & Vallecillo, A. (2002). Quality attributes for COTS components.
13. IBM. Autonomic Computing. http://www.research.ibm.com/autonomic/ (2005).
14. Microsoft. Core Principles of the Dynamic Systems Initiative. http://www.microsoft.com/windowsserversystem/dsi/dsicore.mspx (2005).

15. Worldwide Web Consortium (W3C). Web Services Glossary. http://www.w3.org/TR/ws-gloss/ (2004, February). Velte, A.T. (2010). Cloud Computing: A Practical Approach. McGraw Hill.
16. Endrei, M., Ang, J., Arsanjani, A., Chua, S., Comte, P., Krogdahl, P., Luo, M., & Newling, T. (2004). Patterns: Service-Oriented Architecture and Web Services. IBM Redbooks.
17. Sprott, D., & Wilkes, L. (2004). Understanding Service-Oriented Architecture. http://msdn.microsoft.com/en-us/library/aa480021.aspx. Accessed 1.5.2013.
18. Erl, T., Carlyle B., Pautasso C., & Balasubramanian R. (2012). SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST. The Prentice Hall Service Technology Series from Thomas Erl. Pearson Education.
19. Huifang Li & Cong Chen (2012). A Flexible Workflow Management System Architecture Based on SOA. 2012 International Conference on Affective Computing and Intelligent Interaction: 382-387.
20. Opengroup. Service Oriented Architecture: What Is SOA?. http://www.opengroup.org/soa/source-book/soa_refarch/concepts.htm.
21. ftp://ftp.software.ibm.com/software/soa/pdf/IBMSGMMOverview.pdf.
22. MacKenzie, C.M., Laskey, K., McCabe, F., Brown, P.F., & Metz, R. (2006). Reference model for service oriented architecture 1.0. oasis standard, 12 October 2006. Organization for the Advancement of Structured Information Standards. URL: http://docs. oasis-open. org/soa-rm/v1. 0 (visited on Sept. 13, 2012).
23. Coulouris, G.F., Dollimore, J., & Kindberg, T. (2005). Distributed systems: concepts and design. Pearson Education.
24. McGovern, J., Tyagi, S., Stevens, M. & Mathew, S. (2003). Java Web Services Architecture. Morgan Kaufmann.
25. Oracle, SOA Governance: Framework and Best Practices http://www.oracle.com/us/technologies/soa/oracle-soa-governance-best-practice-066427.pdf.
26. Weill, P. & Ross, J.W. (2004). *IT governance: How top performers manage IT decision rights for superior results.* Harvard Business School Press, Boston, Massachusetts.
27. Cassese, V. (2006). Natural alignment. *Computerworld. 40:* 31-32.
28. Luftman, J. (2004). Managing the information technology resource: Leadership in the information age. Prentice Hall, New Jersey.

29. Bieberstein, N., Bose, S., Fiammante, M., Jones, K. & Shaw, R. (2006). Service-Oriented architecture compass: business value, planning, and enterprise roadmap. IBM Press, Indianapolis, Indiana.
30. Moore, J. (2006). SOA success: five actions you should take. *CIO Insight, 74*, 103-111.
31. Windley, P. (2006). SOA governance: rules of the game. *Info World, 28 (4)*, 29- 35.
32. Latino, R.J., Latino, K.C, & Latino, M.A. (2011). Root Cause Analysis: Improving Performance for Bottom-Line Results. CRC Press.
33. Alonso, G., & Casati, F. (2005). Web services and service-oriented architectures. In Data Engineering. ICDE 2005. Proceedings 21st International Conference on IEEE Web Services and Service Oriented Architectures [Thomas Soddemann, RZG]. Pp. 1147.
34. Web service http://en.wikipedia.org/wiki/Web_service (Last visited 05/07/2010).
35. http://www.infoq.com/articles/SOA-anti-atterns;jsessionid=EA71A2B6A5292 ACC4049A05F7E16BEAD.
36. Crosby, P.B. (1979). Quality is free: the art of making quality certain. McGraw-Hill, New York.
37. Deming, W.E. (1988). Out of the crisis: quality, productivity and competitive position. Cambridge Univ. Press.
38. Feigenbaum, A.V. (1983). Total quality control. McGraw-Hill.
39. Ishikawa, K. (1985). What is total quality control? The Japanese way. Prentice-Hall.
40. Juran, J.M. (1988). Juran's Quality Control Handbook. McGraw-Hill.
41. Shewhart, W.A. (1931). Economic control of quality of manufactured product. Van Nostrand.
42. Brown, W.J., Malveau, R.C., Brown, W.H., McCormick III, H.W. & Mowbray, T.J. (1998). Anti-Patterns: Refactoring Software, Architectures, and Projects in Crisis (1st ed.). John Wiley and Sons.
43. Fenton, N. & Pfleeger, S.L. (1997). Software Metrics: A Rigorous and Practical Approach (2nd ed.). International Thomson Computer Press, London.
44. Fowler, M. (1999). Refactoring – Improving the Design of Existing Code (1st ed). Addison-Wesley.
45. Lewallen, R. (2005). Software Development Life Cycle Models http://codebetter.com/raymondlewallen/2005/07/13/software-development-life-cycle-models/.

46. Ortega, M., Pérez, M., & Rojas T. (2003). Construction of a Systemic Quality Model for evaluating a Software Product. *Software Quality Journal, 11:3*, pp. 219-242.
47. ISO, (1994). ISO 8402:1994 - Quality management and quality assurance - Vocabulary.
48. Kan, S. (2002). Metrics and Models in Software Quality Engineering. Addison-Wesley Longman Publishing Co., Inc. Boston, MA.
49. Mansour, Y.I., & Mustafa, S.H., (2011). Assessing Internal Software Quality Attributes of the Object-Oriented and Service-Oriented Software Development Paradigms: A Comparative Study. *Journal of Software Engineering and Applications*, 4: 244-252.
50. Institute of Electrical and Electronics Engineers, (1990). IEEE 610.12-1990: IEEE Standard Glossary of Software Engineering Terminology.
51. McCall, J., Richards, K., & Walters, F. (1977). Factors in Software Quality. Nat'l Tech.Information Service: Vol. 1, 2 and 3.
52. Boehm, B., Brown, R., Kaspar, H., Lipow, M., McLeod, G., & Merritt, M. (1978). Characteristics of Software Quality. North Holland.
53. Grady, R. & Caswell, D. (1987). Software Metrics: Establishing a Company-Wide Program. Prentice Hall.
54. Dromey, R.G. (1996). Concerning the Chimera [software quality]. IEEE Software, no. 1: 33-43.
55. ISO/IEC 9126-1.2. (1998). ISO/IEC 9126-1.2: Information Technology - Software Product Quality - Part 1: Quality Model, ISO/IEC JTC1/SC7/WG6.
56. Wiegers, K. (2003). Software Requirements (2nd ed.). Microsoft Press.
57. Pettersson, A. (2006). Service-Oriented Architecture (SOA) quality attributes-A research model. MSc thesis, University of Lund.
58. O'Brien, L., Paulo, M. & Len B. (2007). Quality Attributes for Service-Oriented Architectures. In *Proceedings of the International Workshop on Systems Development in SOA Environments SDSOA '07*. IEEE Computer Society, Washington, DC.
59. Peng, Q. (2008). SOA and Quality. MSc thesis, Växjö University.
60. Montagud, S., Abrahao S., & Insfran E., (2012). A systematic review of quality attributes and measures for software product lines. *Software Quality Journal* 20: Issue 3-4: 425-486.
61. Galster, M., Avgeriou, P., & Tofan, D., (2013). Constraints for the design of variability-intensive service-oriented reference architectures – An industrial case study. *Information and Software Technology* 55: 428–441.

62. Marko, M. (2013). Using EBI Pattern in Conjunction with Service-Oriented Architectures. MSc thesis, University of Jyväskylä.
63. IEEE (1998). IEEE Std. 1061-1998, Standard for a Software Quality Metrics Methodology, revision. IEEE Standards Dept., Piscataway, NJ.
64. Jaquith, A. (2007). Security Metrics: Replacing Fear, Uncertainty, and Doubt. Pearson Education Inc., Upper Saddle River, NJ.
65. Burnstein, I. (2003). Practical Software Testing: A Process-Oriented Approach. Springer.
66. Sharble, R. & Cohen, S. (1993). The Object Oriented Brewery: A Comparison of Two object oriented Development Methods. *Software Engineering Notes*, 18, No 2: 60 -73.
67. Chidamber, S. & Kemerer, C. (1994). A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20, No. 6.
68. Henderson-Sellers, B. (1996). Object-Oriented Metrics: Measures of Complexity. Prentice-Hall, New Jersey.
69. Travassos, G., Shull, F., Fredericks, M. & Basili, V. (1999). Detecting Defects in Object Oriented Designs: Using Reading Techniques to Increase Software Quality. Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Denver, Colorado, 1-10.
70. Fenton, N.E. & Neil, M. (2000). Software Metrics: Roadmap. In Finkelstein, A. (ed), Future of Software Engineering. ACM Press.
71. Prnjat, O. & Sacks, L. (2001). Measuring complexity of network and service management components. 2nd IEEE Latin American Network Operations and Management Symposium (LANOMS 2001). Belo Horizonte, Brazil.
72. Bansiya, J. & Davis, C. (2002). A Hierarchical Model for Object-Oriented Design Quality Assessment. *IEEE Transactions on Software Engineering*, 28, No 1.
73. Prechelt, L., Unger, B., Philippsen, M., & Tichy, W. (2003). A controlled experiment on inheritance depth as a cost factor for code maintenance. *The Journal of Systems and Software*, 65: 115-126.
74. Succi, G., Pedrycz, W., Stefanovic, M. & Miller, J. (2003). Practical Assessment of the Models for Identification of Defect-prone Classes in Object-Oriented Commercial Systems Using Design Metrics. *The Journal of Systems and Software*, 65: 1-12.
75. Perepletchikov, M., Ryan, C. & Frampton, K. (2005). Comparing the Impact of Service-Oriented and Object-Oriented Paradigms on the Structural

Properties of Software. In *Second International Workshop on Modeling Inter-Organizational Systems, Cyprus*, 3762: 431-441.

76. Elish, K. & Elish, M. (2008). Predicting Defect-prone Software Modules Using Support Vector Machines. *The Journal of Systems and Software*, 81: 649-660.
77. Jay, G., Hale, J.E., Smith, R.K., Hale, D., Kraft, N.A. & Ward, C. (2009). Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship. *Journal of Software Engineering and Applications*, 2: 137-143.
78. Chowdhury, I. (2009). Using Complexity, Coupling, and Cohesion Metrics as Early Indicators of Vulnerabilities. MSc thesis, Queen's University.
79. Thapaliyal, M. & Verma, G. (2010). Software Defects and Object Oriented Metrics – An Empirical Analysis. *International Journal of Computer Applications* (0975 – 8887), 9, No 5: 1-44.
80. Shaik, A., Reddy, K., Manda, B., Prakashini, C, & Deepthi, K. (2010). An Empirical Validation of Object Oriented Design Metrics in Object Oriented Systems. *Journal of Emerging Trends in Engineering and Applied Sciences (JETEAS)*, 1(2): 216-224.
81. Sanjay, D. & Ajay, R. (2010). A Comprehensive Assessment of Object-Oriented Software Systems Using Metrics Approach. *International Journal on Computer Science and Engineering (IJCSE)*, 2, No 8: 2726-2730.
82. Gurdev, S., Dilbag, S. & Vikram, S. (2011). A Study of Software Metrics. *International Journal of Computational Engineering & Management (IJCEM)*, 11: 22-27.
83. Kehan, G., Taghi, M., Huanjing, W. & Naeem, S. (2011). Choosing software metrics for defect prediction: an investigation on feature selection techniques. *Softw. Pract. Exper*, 41:579–606.
84. Agrawal, D. & Mishra, M. (2012). An Integrated Approach to Measurement Software Defect using Software Matrices. *International Journal of Computer and Organization Trends*, 2 (4): 90-94.
85. Alahmari, S. (2012). A Design Framework for Identifying Optimum Services Using Choreography and Model Transformation. PhD thesis. Faculty of Physical and Applied Science, University of Southampton.
86. Cardoso, J., Mendling, J., Neumann, G. & Reijers H. (2006). A discourse on complexity of process models. In Eder, J. & Dustdar, S. (eds), BPI06 Second International Workshop on Business Process Intelligence in conjunction with BPM 2006. LNCS, 4103: 117-128. Springer-Verlag, Berlin.

87. Papazoglou, M. & Heuvel, W. (2006). Service-oriented design and development methodology. *International Journal of Web Engineering and Technology (IJWET)*, 2(4), 412–442.

88. Abreu, F. & Melo, W. (1996). Evaluating the Impact of Object-Oriented Design on Software Quality. Third International S/W Metrics Symposium, March 1996, pp 1-16. Berlin.

89. Basili, V., Green, S., Laitenberger, O., Lanubile, F., Shull, F., Sorumgard, S. & Zelkowitz, M.V. (1996). The Empirical Investigation of Perspective-Based Reading. *Empirical Software Engineering Journal*, I: 133-164.

90. http://www.isixsigma.com/industries/software-it/defect-prevention-reducing-costs-and-enhancing-quality/.

91. Serena, (2007). An Introduction to Agile a Software Development.

92. Universidad Politecnica de Valencia. Architecture Evaluation Methods: Introduction to ATAM.

93. http://www.isixsigma.com/industries/software-it/software-defect-prevention-nutshell/

94. Lewallen, R. (2005). Software Development Life Cycle Models. http://codebetter.com/raymondlewallen/2005/07/13/software-development-life-cycle-models/.

95. Gueheneuc, Y. (2001). Design defects: A taxonomy. Technical Report INFO-2001, Ecole des Mines de Nantes.

96. Tian, J. (2005). Software Quality Engineering. John Wiley and Sons.

97. Moha, N., Gueheneuc, Y., Duchien, L., & Le Meur, A. (2010). DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering*, 36 (Issue 1): 20-36.

98. Khomh, F., Vaucher, S., Guéhéneuc, Y.-G. & Sahraoui, H. (2009). A Bayesian Approach for the Detection of Code and Design Smells. Proc. of the ICQS 2009.

99. Liu, H.,Yang, L., Niu, Z., Ma, Z. & Shao W. (2009). Facilitating Software Refactoring with Appropriate Resolution Order of Bad Smells. Proc. of the ESEC/FSE 2009. Pp 265–268.

100. Marinescu, R. (2004). Detection Strategies — Metrics-based Rules for Detecting Design Flaws. Proc. of ICM 2004. Pp 350–359.

101. Kessentini, M., Vaucher, S. & Sahraoui, H. (2010). Deviance from Perfection is a Better Criterion than Closeness to Evil When Identifying Risky Code. Proc. of the International Conference on Automated Software Engineering. ASE 2010.

102. Riel, A. J. (1996). Object-Oriented Design Heuristics. Addison-Wesley.

103. Gaffney, J.E. (1981). Metrics in Software Quality Assurance. Proc. of the ACM 1981 Conference. Pp 126-130.

104. Mantyla, M., Vanhanen, J. & Lassenius, C. (2003). A Taxonomy and an Initial Empirical Study of Bad Smells in Code. Proc. of ICSM 2003, IEEE Computer Society.

105. Kothari, S.C., Bishop, L., Sauceda, J. & Daugherty, G., (2004). A Pattern-based Framework for Software Anomaly Detection. *Software Quality Journal*, 12(2): 99–120.

106. Dhambri, K., Sahraoui, H.A. & Poulin, P. (2008). Visual Detection of Design Anomalies. In CSMR, IEEE. Pp 279–283.

107. Erni, K. & Lewerentz, C. (1996). Applying Design Metrics to Object-oriented Frameworks. Proceedings of the 3rd International Software Metrics Symposium, 1996.

108. Alikacem, H. & Sahraoui, H. (2006). Détection d'anomalies utilisant un langage de description de règle de qualité. In actes du 12e colloque LMO.

109. http://www.marouane-kessentini/icpc11.zip.

110. Raedt, D. (1996). Advances in Inductive Logic Programming. IOS Press.

111. Opdyke, W.F. (1992). Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks. PhD thesis, University of Illinois at Urbana-Champaign.

112. Beck, J., & Eichmann, D. (1993). Program and Interface Slicing for Reverse Engineering. In *Proceedings of the International Conference on Software Engineering*, 1: 15-16.

113. Beck, K. (1999). Extreme Programming Explained: Embrace Change. Addison-Wesley, Boston, MA.

114. Beck, K. (2003). Test Driven-Development by Example. Addison-Wesley, Boston, MA.

115. Naouel, M., Yann-Gael, G., Le Meur, A.F. & Laurence, D. (2001). A Domain Analysis to Specify Design Defects and Generate Detection.

116. Nirmal, K.G. & Mukesh, K.R. (2011). An Approach for Detection and Correction of Design Defects in Object-oriented Software. *International Journal of Information Technology and Knowledge Management*, 4(1): 63-67.

117. Florijn, G., Meijers, M. & Winsen, P. V. (1997). To Supply Object-oriented Patterns. Proceedings of ECOOP, 1(2): 5-7.

118. Microsoft (2014). How to: Generate Files from a UML Model", Microsoft 2014: http://msdn.microsoft.com/en-us/library/ff657795.aspx.

119. Aho, A.V., Sethi, R. & Ullman, J.D. (1986). Compilers: principles, techniques, and tools. Addison-Wesley Longman Publishing Co., Inc. Boston, MA.

120. 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-SIGPARSE June 2007, Prague.* Pp 109-120.

121. 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.

122. Rajni Pamnani, Pramila Chawan, Satish Salunkhe. Object Oriented UML Modeling for ATM Systems. Department of computer technology, VJTI University, Mumbai.

123. Wikipedia. "ATM System" www.wikipedia.org/wiki/Automated_teller_machine.

APPENDIX A

Research Tool

INTRODUCTION

Developing quality code is a major concern for the software community. Producing bug-free, extensible, and adaptable code is a hard task. It requires skills, experience, and a deep understanding of the structure and behavior of the software under development. The purpose of the study was to propose a framework to automate the detection of design defects based on design patterns and using design constraints. The framework will offer defects identification and quality measurements in two main steps: **Defects identification based on the design** And **Using the defects to measure and estimate quality.**

This tool examines the relationship between service-oriented architectures (SOAs) and quality attributes. The quality attribute requirements drive software architecture design, it is important to understand how SOAs support these requirements. This tool outlines a set of quality attributes with short definition to describe how those attributes impact the SOA quality and which metrics can help in measuring these impacts.

Position:.....

Experiences in Design Filed: .....

Part (I): Definitions

Design Defects

SOA Design Defects	Definition
Algorithmic and Processing Defects	These occur when the processing steps in the algorithm as described by the pseudo code are incorrect. In the latter case a step may be missing or a step may be duplicated. In the case of algorithm reuse, a designer may have selected an inappropriate algorithm for this problem (it may not work for all cases).
Control, Logic, and Sequence Defects	Control defects occur when logic flow in the pseudo code is not correct. Logic defects usually relate to incorrect use of logic operators, such as less than <, greater than >, etc. These may be used incorrectly in a Boolean expression controlling a branching instruction.
Omission	Necessary information about the system has been omitted from the software artifact.
Incorrect Fact	Some information in the software artifact contradicts information in the requirements document or the general domain knowledge.
Inconsistency	Information within one part of the software artifact is inconsistent with other information in the software artifact.
Ambiguity	Information within the software artifact is ambiguous, i.e. any of a number of interpretations may be derived that should not be the prerogative of the developer doing the implementation.
Extraneous Information	Information is provided that is not needed or used.
Functional Description Defects	The defects in this category include incorrect, missing, and/or unclear design elements. These defects are best detected during a design review.
Intra-class Defects	This category includes any design defect related to the internal structure of a class.

Behavioral Defects	All the design defects related to the application semantics belong to this category.
Inter-class Defects	This category encloses any design defect related to the external structure of the classes (their public interface) and their relationships.

Design Attributes

Design Attributes	Definition
Class	A set of objects that share a common structure and common behavior manifested by a set of methods; the set serves as a template from which objects can be instantiated.
Object	An instantiation of some class which is able to save a state (information) and which offers a number of operations to examine or affect this state.
Method	An operation upon on object, defined as part of the declaration of a class. Methods are operations but not all operations are actual methods declared for a specific class.
Message	A request that an object makes of another object to perform an operation
Design Size	Design size measure the size of design elements, typically by counting the elements contained within. For example, the number of operations in a class, the number of classes in a package, and so on.
Complexity	Complexity measures the degree of connectivity between elements of a design unit. Whereas size counts the elements in a design unit, and coupling the relationships/dependencies leaving the design unit boundary, complexity is concerned with the relationships/dependencies between the elements in the design unit. For instance, counting the number method invocations among the methods within one class can be considered a measure of class complexity, or the number of transitions between the states in a state diagram.
Coupling	Coupling is the degree to which the elements in a design are connected.

Cohesion	Cohesion is the degree to which the elements in a design unit (package, class etc.) are logically related, or "belong together". As such, cohesion is a semantic concept.
Inheritance	A relationship among classes wherein one class shares or methods defined in one (for single inheritance) or more (for multiple inheritance) other classes.
Polymorphism	The ability of an object to interpret a message differently at execution depending upon the super class of the calling object.
Encapsulation	The process of bundling together the elements of an abstraction that constitute its structure and behaviour.
Hierarchies	Hierarchies are used to represent different generalization-specialization concepts in a design.
Abstraction	A measure of the generalization specialization aspect of the design.
Instantiation	The process of creating an instance of the object and binding or adding the specific data.

Metrics

Quality Metrics	Definition
Lines-Of-Code metric (LOC)	Counts the number of statements within a program source code.
Weighted Methods per Class (WMC)	The WMC is a count of the methods implemented within a class or the sum of the complexities of the methods.
Depth of Inheritance Tree (DIT)	It measures the inter-class coupling due to inheritance.
Response set For a Class (RFC)	The RFC is the count of the set of all methods that can be invoked in response to a message to an object of the class or by some method in the class.

Coupling Between Object classes (CBO)	Is a count of the number of other classes to which a class is coupled. It is measured by counting the number of distinct non-inheritance related class hierarchies on which a class depends. It used to compare the level of coupling between classes.
Source Line of Code (SLOC)	The number of executable lines of source code.
Number Of Children (NOC)	The number of children is the number of immediate subclasses subordinate to a class in the hierarchy. It is an indicator of the potential influence a class can have on the design and on the system.
Cyclomatic Complexity (CC)	Is used to evaluate the complexity of an algorithm in a method. It is a count of the number of test cases that are needed to test the method comprehensively.
Lack of Cohesion of Methods (LCOM)	Lack of Cohesion (LCOM) measures the dissimilarity of methods in a class by instance variable or attributes. It defined in terms of the number of pairs of class methods that use common class attributes and the number of pairs of class methods that do not use common class attributes.
Method Inheritance Factor (MIF)	MIF is defined as the ratio of the sum of the inherited Methods in all classes of the system under consideration to the total number of available methods (locally defined plus inherited) for all classes.
Attribute Inheritance Factor (AIF)	AIF is defined as the ratio of the sum of inherited attributes in all classes of the system under consideration to the total number of available attributes (locally defined plus inherited) for all classes.
Polymorphism Factor (PF)	PF is defined as the ratio of the actual number of possible different polymorphic situation for class Ci to the maximum number of possible distinct polymorphic situations for class Ci.
Method Hiding Factor (MHF)	MHF is defined as the ratio of the sum of the invisibilities of all methods defined in all classes to the total number of methods defined in the system under consideration. The invisibility of a method is the percentage of the total classes from which this method is not visible.

Attribute Hiding Factor (AHF)	AHF is defined as the ratio of the sum of the invisibilities of all attributes defined in all classes to the total number of attributes defined in the system under consideration.

Quality Attributes

SOA Quality Attributes	Definition
Adaptability	The quality of being adaptable to changes. The use of an SOA approach should have a positive impact on adaptability, as long as the adaptations are managed properly. However, the management of this quality attribute is left up to the service users and providers, and no standards exist to support it. This attribute must be managed in coordination with other quality attributes including stability, performance, and availability, and the necessary trade-offs must be identified and made.
Analysability	The capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified.
Auditability	Auditability is the quality factor representing the degree to which an application or component keeps sufficiently adequate records to support one or more specified financial or legal audits. With the ever-increasing need for systems to comply with business and regulatory legislation (financial and health sectors especially), the ability to audit a system for compliance is an important consideration.
Availability	Availability refers to the ability of the user community to access the service, whether to submit new request, update or alter existing request, or collect the results of previous request.
Changeability	The capability of the software product to enable a specified modification to be implemented.
Correctness	Accountability for satisfying all requirements of the system. Measure of exact adherence to specifications.

Extensibility	Extending an SOA by adding new services or incorporating additional capabilities into existing services is supported within an SOA. However, the interface/formal contract must be designed carefully to make sure that it can be extended, if necessary, without causing a major impact on the service users.
Interoperability	The ability to exchange and use information (usually in a large heterogeneous network made up of several local area networks). Interoperability may occur between two (or more) entities that are related to one another in one of three ways: **Integrated:** where there is a standard format for all constituent systems **Unified:** where there is a common meta-level structure across constituent models, providing a means for establishing semantic equivalence **Federated:** where models must be dynamically accommodated rather than having a predetermined meta-model.
Modifiability	Modifiability considers how the system can accommodate anticipated and unanticipated changes and is largely a measure of how changes can be made locally, with little ripple effect on the system at large. The world around most Software System is constantly changing. This requires software systems to be modified several times after their initial development.
Performance	Performance refers to the system responsiveness: either the time required responding to specific events, or the number of events processed in a given time interval. An SOA approach can have a negative impact on the performance of an application due to network delays, the overhead of looking up services in a directory, and the overhead caused by intermediaries that handle communication. The service user must design and evaluate the architecture carefully, and the service provider must design and evaluate its services carefully to make sure that the necessary performance requirements are met.
Reusability	The degree to which a software module or other work product can be used in more than one computing program or software system.

Scalability	Scalability is the ability of SOA to function well when the system is changed in size or in volume in order to meet users' needs. Extending an SOA by adding new services or incorporating additional capabilities into existing services is supported within an SOA. However, the interface/formal contract must be designed carefully to make sure that it can be extended, if necessary, without causing a major impact on the service users.
Security	Due to the distributed nature of the current enterprise systems, we have difficulty in administering security policies and bridging diverse security models. This leads to increased opportunities to make mistakes and leave security holes; hence the chance of accidental disclosure and the vulnerability to attack goes up.
Stability	The capability of the software product to avoid unexpected effects from modifications of the software.
Testability	Testability can be negatively impacted when using an SOA due to the complexity of the testing services that are distributed across a network. Those services might be provided by external organizations where access to the source code is not available, and if they implement runtime discovery of services, it may be impossible to identify which services are used until a system executes. It is up to the service users and providers to test the services, and very little support is currently provided for the end-to-end testing of an SOA.
Understandability	The degree to which the purpose of the system or component is clear to the evaluator.
Usability	Usability may decrease if the services within the application support human interactions with the system and there are performance problems with the services. It is up to the services users and providers to build support for usability into their systems.
Maintainability	The ability to identify and fix a fault within a software component is what the maintainability characteristic addresses.
Reliability	The capability of the system to maintain its service provision under defined conditions for defined periods of time.

Part (II): Selection Process

SOA Design Attribute	Metrics	SOA Design Defects	SOA Quality Attributes																			
			Adaptability	Analysability	Auditability	Availability	Changeability	Correctness	Extensibility	Interoperability	Modifiability	Performance	Reusability	Scalability	Security	Stability	Testability	Understandability	Usability	Maintainability	Reliability
Class	LOC	Algorithmic and Processing Defects																			
Object	WMC																				
Method	DIT	Control, Logic, and Sequence Defects																			
Message	RFC																				
Design Size	CBO	Omission																			
Complexity	SLOC	Incorrect Fact																			
Coupling	NOC	Inconsistency																			
Cohesion	CC	Ambiguity																			
Inheritance	LCOM	Extraneous Information																			

Polymorphism	**MIF**	**Functional Description Defects**																				
Encapsulation	**AIF**																					
Hierarchies	**PF**	**Intra-class Defects**																				
Abstraction	**MHF**	**Behavioral Defects**																				
Instantiation	**AHF**	**Inter-class Defects**																				

Part (III): Metrics Measurement Range

Impact On Quality [(10) Highest impact – (1) lowest impact]

Quality Metrics	SOA Quality Attributes																		
	Adaptability	Analysability	Auditability	Availability	Changeability	Correctness	Extensibility	Interoperability	Modifiability	Performance	Reusability	Scalability	Security	Stability	Testability	Understandability	Usability	Maintainability	Reliability
LOC																			
WMC																			
DIT																			
RFC																			
CBO																			
SLOC																			
NOC																			
CC																			
DIT																			
LCOM																			
MIF																			
AIF																			
PF																			
MHF																			
LOC																			

SOA Quality Attributes Weights

SOA Quality Attributes	Adaptability	Analysability	Auditability	Availability	Changeability	Correctness	Extensibility	Interoperability	Modifiability	Performance	Reusability	Scalability	Security	Stability	Testability	Understandability	Usability	Maintainability	Reliability	Total
Weights (Out of 100)																				100

Part (IV): Selection Process

SOA Design Attribute	Metrics	SOA Design Defects	SOA Quality Attributes																			
			Adaptability	Analysability	Auditability	Availability	Changeability	Correctness	Extensibility	Interoperability	Modifiability	Performance	Reusability	Scalability	Security	Stability	Testability	Understandability	Usability	Maintainability	Reliability
	LOC	Algorithmic and Processing Defects																			
	WMC																				
	DIT	Control, Logic, and Sequence Defects																			
	RFC																				
Design Size	CBO	Omission																			
Complexity	SLOC	Incorrect Fact																			
Coupling	NOC	Inconsistency																			
Cohesion	CC	Ambiguity																			
Inheritance	LCOM	Extraneous Information																			

Polymorphism	**MIF**	**Functional Description Defects**																				
Encapsulation	**AIF**																					
Hierarchies	**PF**	**Intra-class Defects**																				
Abstraction	**MHF**	**Behavioral Defects**																				
Instantiation	**AHF**	**Inter-class Defects**																				

Part (VI): Metrics Measurement Range

Impact On Quality [(10) Highest impact – (1) lowest impact]

Quality Metrics	SOA Quality Attributes																		
	Adaptability	Analysability	Auditability	Availability	Changeability	Correctness	Extensibility	Interoperability	Modifiability	Performance	Reusability	Scalability	Security	Stability	Testability	Understandability	Usability	Maintainability	Reliability
LOC																			
WMC																			
DIT																			
RFC																			
CBO																			
SLOC																			
NOC																			
CC																			
DIT																			
LCOM																			
MIF																			
AIF																			
PF																			
MHF																			
LOC																			

SOA Quality Attributes Weights

SOA Quality Attributes	Adaptability	Analysability	Auditability	Availability	Changeability	Correctness	Extensibility	Interoperability	Modifiability	Performance	Reusability	Scalability	Security	Stability	Testability	Understandability	Usability	Maintainability	Reliability	Total
Weights (Out of 100)																				100