A Contracts-Based Model for Managing Web Services Evolution

by

Chiponga Kudzai

(201329712)

A dissertation submitted in fulfilment of the requirements for the degree of

Master of Science in Computer Science

in the Faculty of Science and Agriculture

Department of Computer Science

University of Zululand

KwaDlangezwa

Supervisor: Prof. M.O Adigun

Co-supervisor: Mr. P. Tarwireyi

2015

ABSTRACT

Service-based systems need to be designed in such a way that they are able to accommodate the volatility of the environment in which they operate. Failure to evolve service-based systems will result in service providers losing their competitive edge. Additionally, failure to evolve these systems properly will have far-reaching negative impacts on all stakeholders, especially if disruptions are allowed to occur. However, evolving service-based systems in a non-disruptive manner is still a challenge both to the research community and industry. Over recent years, many organisations have been adopting technology-based service solutions in what has become a technology-driven business environment, with web services at the forefront of being a business-enabler. This dissertation focusses on developing a contracts-based model for managing the evolution of web services in a manner that is consistent and transparent to business partners. The design science research methodology was used to architect a model that can alleviate the challenge of evolving service oriented systems. It was found that technical web service contracts can be leveraged upon to manage and maintain consumers while evolution is carried out. A contracts-based service proxy was developed as an instantiation of the model. Experiments demonstrated that this proxy maintained compatibility between evolving services and existing consumers. The model developed in this research presented a cost effective solution to managing the evolution of webservices while reusing the same computing resources and cutting down on the development time needed in evolving web services. Even though the proxy introduced processing overheads, the resultant loss in service throughput was negligible, especially when we take into consideration the amount of time and effort taken in evolving services manually.

DECLARATION

I, Kudzai Chiponga, declare that this dissertation represents my own work and that this work has not been previously submitted to any university or institution of tertiary education. All sources of information used in this work have been acknowledged.

Kudzai Chiponga

Signature……

Date ……

DEDICATION

I dedicate this work to the Author and Finisher of our Faith.

ACKNOWLEGEMENTS.

I would like to thank my supervisors Professor Matthew. O. Adigun and Mr. Paul. Tarwireyi, for their expertise, support and mentorship, without which, this work would not have made it to fruition. Your continued dedication to both me and my work can never be equalled. I look forward to more fruitful work with you again beyond this point.

A special thank you goes to Brother Edgar Jembere, for his invaluable contribution in the reasoning around this work. I would also like to extend my thanks to my colleagues and friends not in any order of importance, namely, Z. Nxumalo, S. Dlamini, T. Akinola, M. Shabalala, D. Afuro, S. Fakude, L. Qwabe, O.Oki, N. Sibeko, B. Mutanga, P. Mudali, S. Fatyi, N. Mdletshe, T. Ntuili, D. Zibani, Professor S.S.Xulu, I. Mba, S. Mthembu, O.J. Pooe, S. Makumire, X. Makoba and U. Tsuro for the continued support and motivation to soldier on.

A thank you again goes to my family for their unwavering support and love, and for bearing with my absence throughout the duration of this work.

Last and most importantly, I would like to extend my sincere gratitude to Dr. T. Chirima, my pillar and source of strength.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF EQUATIONS

LIST OF PUBLICATIONS

Chiponga, K., Tarwireyi, P., Adigun, M.O., "Contract-based Web Service Evolution Model", In: *2014 Proceedings of the Southern Africa Telecommunication Networks and Applications Conference (SATNAC)*, Boardwalk Conference Centre, Nelson Mandela Bay, Eastern Cape, South Africa. 2014

Chiponga, K., Tarwireyi, P., Adigun, M.O., "A Version-Based Transformation Proxy For Service Evolution", In: *The 6th IEEE International Conference on Adaptive Science and Technology (ICAST) 2014,* Covenant University, Ota, Nigeria. 2014

CHAPTER ONE

1. INTRODUCTION

1.1. Research Context and Motivation

In today's business world, effective marketing is the key to business growth. Internet Service Providers (ISPs) and website hosting companies have seen the demand for web hosting space increase over the years. The increase in demand for hosting space is because organisations are turning to websites as a marketing tool. Having a solid Internet presence is therefore critical to business success. Internet presence may be mistaken, by some, to refer to an organisation having a good website only, however an Internet vantage point goes well beyond that. It extends to the ability of an organisation to consume services offered by partners and or other organisations (as third-parties in a business process) to enhance business operations in delivering value for their customers. Businesses are adopting Service Oriented Computing as a business enabler.

Service Oriented Computing (SOC) is a relatively new computing paradigm within Services Oriented Architecture (SOA) which exhibits characteristics that favour automation of some complex business processes over a network across a variety of stakeholders. The characteristics of SOC include but are not limited to: the ability of services to be published, discovered, composed and bound to; platform-independence; loose-coupling; and self-contained interactions enabling independence of any interaction from others (Papazoglou, 2008a). SOC allows for services to interoperate between heterogeneous systems integrating complex systems to reduce the expense of rebuilding and rewriting pieces of code for every new SOA application being built. The generic SOA is composed of three main functionalities, which are the service provider, the service consumer and the registry (Papazoglou, 2008a).

Businesses tapping into the nature of SOC harness the advantages of a SOA implementation as it is a cost effective solution to enhancing old systems that were not designed with the services architecture in mind (hereafter referred to as legacy systems). A SOA implementation may be realised in the form of web services, which can be composed of other web services giving rise to a web service ecosystem. Due to the distributed nature of a SOA implementation (Tsai *et al.*, 2002), these web services used in the composite web service are not owned by one organisation, and are not housed under one roof. Not having these web services under one roof and one administration ushers in a new set of challenges in administering systems. The challenges emanate from the fact that one organisation cannot control the internal systems of another organisation.

A typical SOA implementation has three main entities, namely: a service provider, a service registry and at least one service consumer (Bloomberg, 2015). The entity considered as a service provider is responsible for the development, deployment and registration of a web service. The implementation detail lies with the service provider and is not necessarily to be revealed to the outside world. Only the information about how to interact with the web service and what response to expect from the web service are made publicly available through the process of publishing that information in a service registry. The service registry is an eXtensible Markup Language (XML) based registry known as the Universal Description, Discovery and Integration (UDDI) (Mallayya *et al.*, 2015). The service registry may be a public or a private UDDI depending on who the service provider targets as the end users of their service offerings. The information published in the registry is the description of the web service written using XML in a file known as a Web Services Definition Language (WSDL) file for Service Oriented Architecture Protocol (SOAP) based web services. The WSDL file contains information such as the location of the web service, how to invoke/use the service, the parameters to pass to the web service in order to invoke it and the

expected response or result that the web service sends back to the service consumer. The WSDL offers all the control information needed by a customer in order to know how the service can be used and what to expect from the service. This makes the WSDL an essential part of the service capable of being leveraged for more than just service description. The service consumer is interchangeably referred to as the client to a web service, or consumer, the term consumer will be used throughout this work. The consumer is usually in the form of a program implementation by the persons or organisation seeking to use a service provided in the form of a web service. The consumer is built based on a particular WSDL file retrieved from a registry in order to conform to the expected methods and parameters of invocation of the web service. A consumer can be the end-system, implying that the consumer displays the response from the web service or uses the returned result for further computation and execution of the functions the consumer was designed for. Consumers may also be service providers to other service consumers. Thus, a web service can be composed of other web services in order to orchestrate a complete business transaction, resulting in an ecosystem of web services.

Web services are used in general to perform a business function for users or other software applications. In a typical SOA implementation, the services are not limited to a single user, hence, a web service is designed for a set of users sharing some common functionality provided by the web service. However, the service providers do not explicitly know the set of users who will use their service. The users find the WSDL file in a registry and build consumers to bind to the web service without the need to inform and formally signup with the service provider. As businesses operate, their expectations of the service offerings are not entirely the same yet still rely on the same service for the execution of a particular function. The different expectations of the web service consumers influence the web services to be changed in order to meet the requirements of

other users, while at the same time trying to maintain the web service for the users who are satisfied with the current web service. An ever-changing and competitive business environment implies that software will change and in turn, influences rapid changes in deployed web services, inevitably affecting the web services ecosystem. As technology advances, service interfaces, data types, operations, expected messages and exchange patterns, can change as web services evolve to meet changing business rules and changing operating environments. Some of these changes in a web service may cause disruption to the interaction between a service provider and a service consumer, resulting in a possible chain of operational and business losses to the organisations relying on the web service. It is therefore of paramount importance that these changes be implemented in a controlled manner so as to minimise, if not eradicate business and service losses.

Studies on software evolution have been ongoing since the late 1960s, leading up to the birth of a series of Laws now commonly referred to as the Laws of software evolution. These Laws are also known as Lehman's Laws of software evolution formulated around 1974 (González-Barahona *et al.*, 2014). Around 1980, Lehman further classified programs into the SPE-scheme where:

- S-type programs do not evolve and are written from a static specification
- P-type programs are a mixture of S and E-type programs in which the specifications cannot be completely defined before the program exists and
- E-type programs are a reflection of human processes or in SOA terms, business process modelling

The process of E-type software evolution has proved to be a challenge owing to their multi-input and multi-output nature. In a typical SOA implementation, there may be limited to no feedback from targeted customers during the development or the evolution of a web service. Some of

Lehman's Laws applied in E-type systems also apply in SOA, as a SOA implementation attempts to address business problems in the real world environment.

1.2. Problem Statement

Given that SOC adoption is quite recent (Kijas and Zalewski, 2013), much of the research effort has been focussed on the building and deployment of web services. From an engineering perspective, processes to support the evolution of service oriented systems are a challenge (Lewis *et al.*, 2010). Conventional development methodologies such as Object-Oriented Analysis and Design (OOAD), Component-Based Development (CBD) and business process modelling, despite their usefulness, do not address the key elements of SOC (Papazoglou, 2008a). Conventional development methodologies can only address some of the requirements of SOC applications. Unlike traditional software, Services are not necessarily owned by a single organisation and are consumed by more than a single unknown and randomly developed consumer. As a result, service providers may not know how many clients use their services or how often they are used by the clients (Fokaefs *et al.*, 2011).

Service evolution precedes successful service adaptation (Bellahsène and Léonard, 2008). The current assumption with service adaptation approaches is that services can evolve independently of other services yet when a change is made, service consumers are not immediately aware of the service change made. Services will inevitably change over time to meet ever changing business requirements (Papazoglou, 2008a). These changes and modifications result in altered services (Andrikopoulos *et al.*, 2012), which if uncontrolled will disrupt the operations of consumers that depend on the services. Hence changes introduced at the service provider level may cause severe disruptions on the client's side (Bellahsène and Léonard, 2008). This was evident in a study to analyse evolution practices employed by large API providers such as Google Maps, Facebook,

Twitter and Netflix (Espinha *et al.*, 2015). According to Espinha *et al.,* (2015), a survey entitled "API integration pain", conducted among 130 Application Programming Interfaces (API) client developers, showed there are a lot of complaints about current web service providers. The developers claimed that the APIs randomly change without warning, causing their systems to break. Developers mentioned that they are faced with an endless struggle to keep up with the changes pushed by the web service providers. They said: *"... As developers we build our livelihoods on these APIs, and we deserve better"*. In a subsequent survey, Facebook came out as the worst because of its never-ending breaking API changes (Perez, 2015).

In order therefore to manage changes in a meaningful and effective manner, the service consumers using a service that needs to be upgraded by the client developers must also be considered when service changes are introduced on the service provider's side (Bellahsène and Léonard, 2008). Failure to do so will most certainly result in severe application disruption (Kijas and Zalewski, 2013). Unfortunately, the management of change in the context of service orientated computing environments has, to the best of the researcher's knowledge, not been discussed sufficiently thus far. As supported by (Kontogiannis *et al.*, 2008; Lewis and Smith, 2013), not much research specifically addresses or provides guidelines for maintenance activities in a service-oriented environment. Hence, there is a need for a methodology for evolving shared services which will help identify users of the service, how they will be affected, and estimate the potential business costs when changes cause disruptions (Fokaefs *et al.*, 2011; Lewis and Smith, 2013). The methodology should evolve shared services while maintaining consistency (well-formed and valid product) as the services evolve from one state to the next. This view has driven the formulation of the following research questions.

1.3. Research Questions

How can service contracts be used to incorporate and manage service evolution?

Sub-Research Questions:

1. What is the state of the art of service evolution in SOC and what other fields can we take lessons from?
2. How can we design a model to ensure that there will be no major disruptions to business functions after a service is upgraded or changed?
3. How can we identify the service version that is being requested by a consumer?
4. How can the proposed model be validated and what mechanisms or procedures can be used to evaluate the efficacy and utility of the proposed solution?

1.4. Research Goal and objectives

To tackle the research problem that has been highlighted, this research work's main goal is to:

- Formulate a contract-based model for managing evolution of shared services in order to minimise disruptions to service consumers when a service changes.

1.4.1. Research Objectives

To achieve this goal, the objectives that need to be met are to:

- Establish how the WSDL file (also known as the contract) can be incorporated into the management of services evolution
- Develop a mechanism for identifying which service version is being requested by a consumer

- Develop a service message transformation proxy relying on contracts, upon which tests and assessments of the proposed model will be conducted

1.5. Research Methodology

In fulfilling the goal of this research, the design science methodology was followed, where a given problem is solved by introducing new artifacts into the environment (Hevner and Chatterjee, 2010). Design science seeks to come up with new and innovative artifacts to enhance human and organisational capabilities through the use of Information Systems (IS). Artifacts include constructs, models, methods and instantiations but not explicitly the process by which such artifacts evolve (Göbel and Cronholm, 2012; Hevner and Chatterjee, 2010). These artifacts help us to understand the problem being addressed and the feasibility of the artifacts' approach to the solution.

In this research, the goal was to develop a contracts-based model for managing services evolution. This model is the artifact of this research. However, when following design science, it is not enough just to produce an artifact, the artifact has to be evaluated to determine its feasibility. In this research, the model was instantiated into a prototype that provided proof-by-construction.

The research methodology in this work was broken down into four phases namely: the problem analysis phase, establishment of the state of the art phase, solution design, and the evaluation phase.

1.5.1. Phase 1. Problem analysis

According to the design science methodology, the first step when conducting research is to understand the problem at hand. This understanding will help construct artifacts aimed at changing the phenomena (Hevner *et al.*, 2004). The research context and motivation helped in finding the

problem statement stated in Section 1.2. The problem statement was broken down into the research questions, following which a literature review was conducted.

1.5.2. Phase 2. Establishment of the state of the art

A state of the art analysis of literature was conducted to ascertain what has been done to address services evolution concerns in SOC. It is very important to analyse and understand the relevance of any research work to the domain in which it is being carried out. This helped identify the problem in the literature and establish to what extent contracts have been used to manage services. This phase in the methodology aided in achieving the first objective which was to establish how web service contracts can be used in managing service evolution. The combination of phase 1 and phase 2 helped to answer the 1st research sub-question.

1.5.3. Phase 3. Solution design and implementation

The result of design science is a purposeful Information Technology (IT) artifact(s) designed to solve an important problem in industry (Hevner *et al.*, 2004). This phase involved the design and the implementation of the solution. After this solution was designed as a response to the 2nd research sub-question, a service message transformation proxy which relies on service contracts was implemented as a proof-of-concept prototype. The prototype uses a common use-case SOA application described in Chapter 3, Section 3.1 and 3.3 of this dissertation, that other scholars proposed as a standard case study which researchers can use to compare and benchmark their solutions (Guo *et al.*, 2011; Tiago Espinha, 2012). How useful the proposed solution is and in what ways this solution can be adopted in industry was demonstrated using the prototype with the aid of a running scenario. As part of the solution, a proxy that could match and identify compatible

consumers to web service implementations was implemented, thus achieving the second research objective. This helped in answering the third research sub-question.

1.5.4. Phase 4. Evaluations

This is one of the most significant tasks in design science. The utility, quality and efficiency of an artifact must be demonstrated via well-executed evaluation methods (Hevner *et al.*, 2004). Performance tests were carried out to measure the quality of service parameters which needed to be maintained during the evolution of a web service. This assisted in achieving the third research objective. The checking of consistency of the service before and after evolutionary processes were governed by the Laws of evolution and the best practices in software configuration management. Evaluation methods in design science also include:

- Testing: The proposed solution was examined to see where it fits in the state of the art
- Performance tests: The efficiency of the proposed model was tested to find the request/response times and efficiency of change-over times for evolving web services
- Experimentation: Simulation and execution of the artifacts was performed with artificial data

The collected and analysed data enabled the defining and refining of the model that was designed for managing the evolution of web services. This phase was in an effort to answer the fourth research sub-question.

1.6. Delimitations

Evolution and maintenance of web services still have challenges that this research work does not address. It is therefore imperative that this research work was done with some limitations and assumptions having been made. Some of the assumptions and limitations are listed below:

- Service evolution dictates that there should be a trace of evolution between new services and their predecessors. Therefore it is important that the implementation of new services should always show the link or the relationship with the old services, otherwise it would be regarded as a new product. The model therefore assumes that there are similarities between subsequent service versions
- This research assumes that the service registry is implemented within the Enterprise Service Bus (ESB) that was selected and that all contract versions are published and retrieved from this registry
- This work was simulated to resemble a distributed architecture, however this architecture was not across the Internet but in a laboratory setting. Hence, the traffic results would not reflect a scenario wherein other forms of network traffic exist other than the SOAP requests and responses generated by the simulation

1.7. Structure of Dissertation

The remainder of this work is structured as follows:

Chapter 2 presents a review of the work that has been done in addressing some of the concerns in web services evolution and brings to light some lessons that can be learnt from other disciplines in IT that can be modified and adapted to web service evolution. Chapter 3 gives a description of the running scenario around which the research method was applied. This is the running scenario that is referred to thereafter in the chapters that follow. Chapter 4 presents the model that was proposed in managing the evolution of a web service and the design criteria behind this model. The validation of the model, and the evaluation thereof, are presented in Chapters 5 and 6 respectively, to show the technical feasibility, and applicability of the contracts-based proxy

model. The economic implications of the contracts-based proxy approach are also discussed in Chapter 6. Chapter 7 concludes the dissertation and summarises the findings from the work, and the possible future directions to extending and improving the work are also briefly discussed.

CHAPTER TWO

2. BACKGROUND AND STATE OF THE ART ANALYSIS

This chapter aims to achieve the first objective stated in Section 1.5 of Chapter 1, which is to establish how contracts can be incorporated into the management of service evolution. Establishment of how contracts can be employed in service evolution could enable us to determine the mechanisms to be developed as stated in the remaining two objectives of this work. This analysis helps us respond to the first research sub-question by picking out and drawing lessons from other disciplines in IT that can be adopted and adapted in addressing some of the evolution concerns in web services.

In step with the chosen research methodology as established in Chapter 1, Section 1.5, design science research methodology phase 2, is the problem identification phase in which the gap that exists in the work and accomplished so far was identified, to the best knowledge of the author.

2.1. Web services and SOA

Over the years, services have become the building blocks of distributed applications and systems (Gu and Lago, 2011). One of the main reasons why services are now a more preferred option for distributed systems than networked legacy systems is because they offer a common interface and a common set of standards that enable interoperability across organisations that need to exchange information (Berners-Lee, 2009). To complete a business task, businesses offer each other services that perform subtasks of a business transaction and this is referred to as a business process. A common interface that is publicly available means that more businesses can integrate and share resources to complete a business process. This also brings competition and flexibility in choosing whose services to use, thus opening up a whole world of possibilities in service integration and

service delivery. SOC facilitates the interoperability of the services across organisations and across different software and hardware platforms, removing the barriers to information exchange in this networked and growing worldwide online economy (Bhuvaneswari and Sujatha, 2011).

The maintenance activities and evolution in SOC systems are made complex and challenging largely due to the distributed nature of SOA (Hägg *et al.*, 1996). The services comprising a distributed system may not only reside on different servers in an organisation but may be across organisations, placing the administration of some parts beyond the control of any particular organisation (Fokaefs and Stroulia, 2012). Due to enterprise competition, the key to remaining competitive lies in an organisation's agility to respond to a fluid competitive environment susceptible to an ever changing set of stakeholder requirements and expectations (Govardhan and Feuerlicht, 2009). Responding to changing requirements and being proactive or innovative on the part of any service provider means that the services they offer would have to change to meet the expectations of their service requestors.

SOA is described as a business enabler providing easy composition of distributed applications, software reuse, business agility and rapid and low-cost development (Bloomberg, 2015). SOA is an architectural approach to addressing problems of integrating different enterprise legacy systems in closed environments and minimizing the dependencies between them. In reality, these systems would have been developed at different times, by different people or organisations for different reasons using different technologies and programming platforms and languages and for varying sets of end-users (Khadka *et al.*, 2013). SOA enables these systems to interconnect and share information across the Internet in a standard way understood by humans and more importantly, by machines. A SOA implementation being platform and language independent implies that different systems interacting through the Internet via messaging can be grouped together to complete

complex business processes, thereby allowing for extensive software reuse in a now distributed environment (Lewis *et al.*, 2005). SOA implementations can thus use other available services online without having to develop new in-house applications to accomplish the same business process and this speeds up development-time and cuts down on development budgets (Ren and Lyytinen, 2008). SOA enables reuse of existing platforms and services thus emphasising business value as opposed to the technologies underlying the implementation thereof.

The basic Service Oriented Architecture depicted in Figure 2.1 has three components (Registry/Discovery Agents, Provider, and Consumer/Requestor) which are related by a set of operations, commonly summarised as the Publish, Find and Bind/Interact operations.

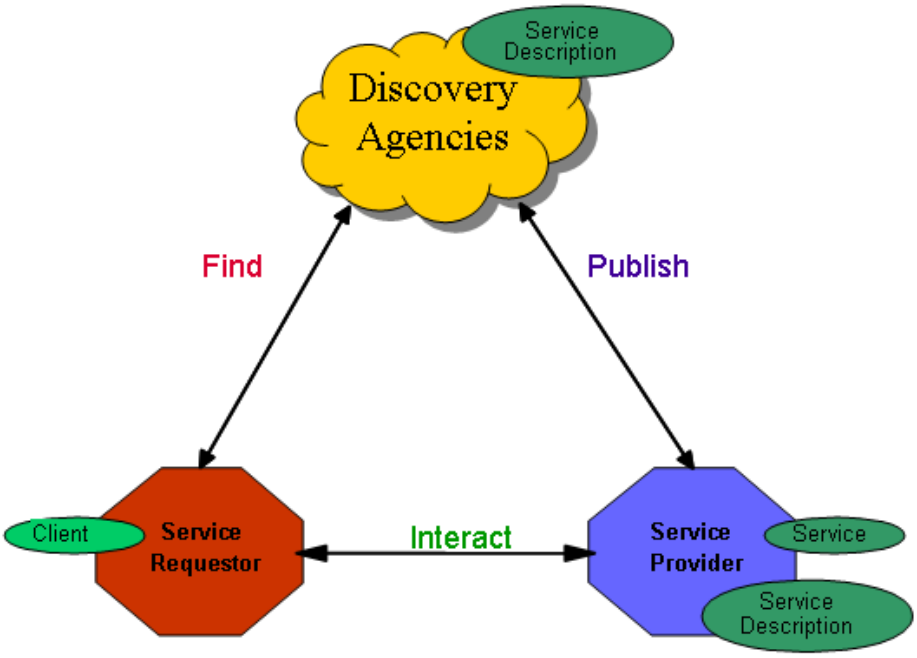

Figure 2.1: The Basic Service Oriented Architecture ("Web Services Architecture," n.d.)

Following the basic service oriented architecture depicted in Figure 2.1, Figure 2.2 shows the relationships between the entities involved. For example, Figure 2.2 shows that a person or organisation provides or owns a service. Another person or organisation owns a consumer which

uses the service and the consumer is a service requestor agent. The service has a service description which is housed or published in discovery agencies. The service description defines the service interface and the messages that need to be passed in order for a consumer to use the service. The service executes an action, resulting in a response message that is returned to the consumer.

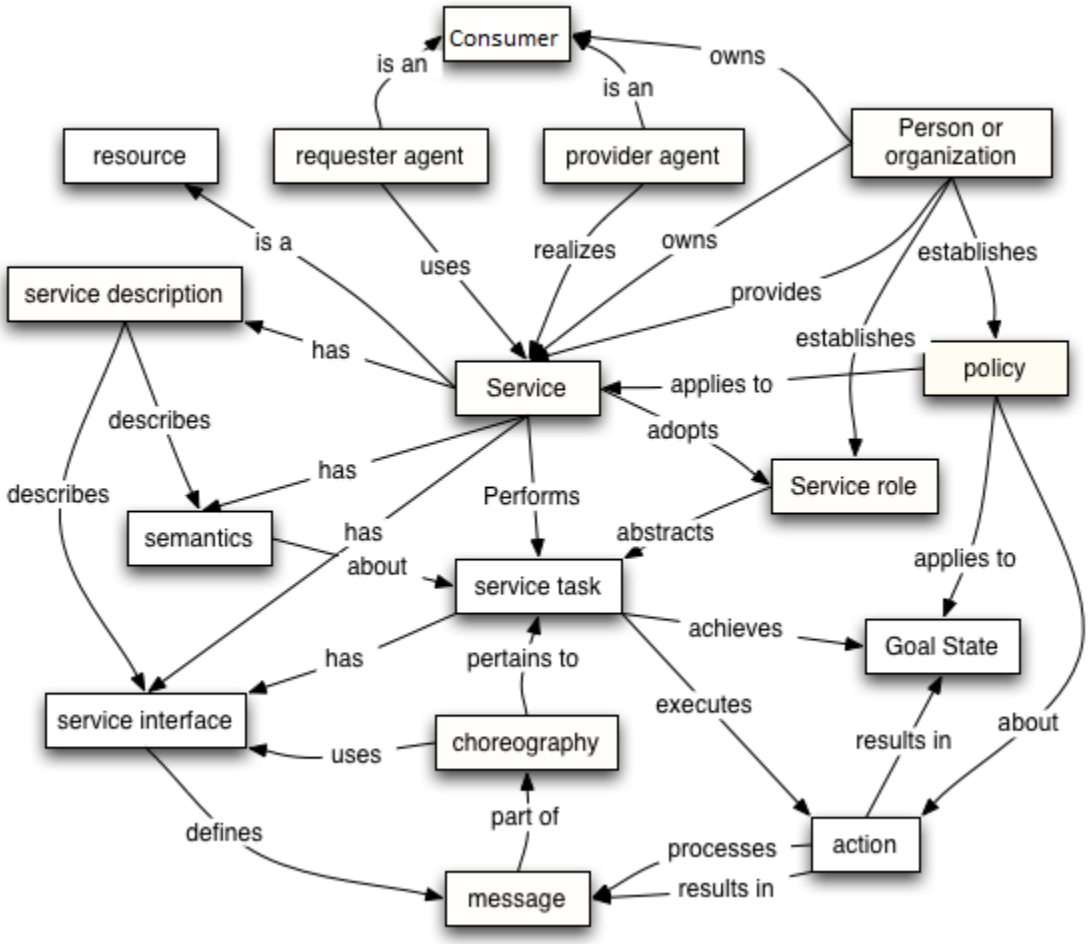

Figure 2.2: The Service Oriented Model (Booth et al., 2004)

A typical implementation of a SOA can be in the form of a web service. A web service can thus be described as any piece of software that makes itself available over the Internet and uses a standardized XML messaging format (Alonso *et al.*, 2004; Lippert and Govindarajulu, 2015). There is no one specific definition of what exactly a web service is as this term may mean different

things to different people (Berners-Lee, 2009), and various definitions have been given. A web service can be summarised as being any service exhibiting the following characteristics:

- Is not tied to any operating system or programming language
- Can be invoked over the Internet
- Is self-describing
- Can be discovered online

There are two kinds of web services, Representational State Transfer (REST) and Simple Object Access Protocol (SOAP) (Khan and Abbasi, 2015; Mulligan and Gracanin, 2009). These present a method of communication mainly between devices over the Internet. SOAP defines a set of rules for the XML messages that are exchanged between systems using various protocols, such as Send-Mail Transfer Protocol (SMTP) and Hypertext Transfer Protocol (HTTP). Since HTTP and SMTP are unblocked on most firewalls and used as standard ports, SOAP needs no special reconfigurations to tunnel across firewalls.

On the other hand, REST presents an architectural style in which each object is represented by a unique Uniform Resource Locator (URL). REST uses the POST, GET, PUT, DELETE, and HEAD standard operations in HTTP to manipulate the contents of the objects. REST is considered lightweight and easy to build whereas SOAP is easy to consume and adheres to a contract.

For simple smaller Application Program Interfaces (API), and faster results with a lower learning curve, REST would be ideal, but for more complex systems that need to be exposed to the outside world, SOAP is more useful as it is regulated by standards that are globally accepted (Mulligan and Gracanin, 2009; Wagh and Thool, 2012).

Following the Service Oriented Architecture, although consumers bind to the service, the service provider has no full knowledge of all the consumers using the service, and this brings a new set of service maintenance and evolution challenges (Kajko-Mattsson *et al.*, 2008). An improperly planned maintenance activity on a service or an update that takes a service from a service state from up to down or a service change that changes the structure of the messages to be exchanged between systems. These are just a few examples of maintenance and evolution activities that can have far reaching and potentially disastrous consequences for the consumers of that service (Kajko-Mattsson *et al.*, 2008).

2.2. SOAP Web services

SOAP is an XML-based protocol that is used to exchange information among web services. SOAP was developed to address the limitations of the conventional distributed communication protocols such as DCOM, Java/RMI and other application-to-application across-the-Internet communications. SOAP facilitates interoperability in a distributed architecture and is platform independent, making it accessible and available to a wide range of programs. SOAP messages can be transported across the Internet via a number of protocols, for example, HTTP, SMTP and Java Message Service (JMS) (Papazoglou, 2008a).

SOAP embodies the following advantages which make it popular with web services (Tsalgatidou and Pilioura, 2002):

- SOAP is based on XML, making it simple and easy to parse
- SOAP easily passes through firewalls without the need for firewall reconfigurations as it uses HTTP / SMTP whose ports are permitted on most if not all firewalls

- SOAP uses open standards. The use of STANDARDS implies that it conforms to some set of rules making it extendable and easy for the community to support
- SOAP is the most widely accepted standard and is portable. It can be used on any platform from high capacity server machines to limited profile devices

SOAP web services are described using the WSDL, which defines the available functions and how information as SOAP messages can be transferred from the requestor to the service provider and vice-versa. WSDL was standardised by the World Wide Web Consortium (W3C) and has wide acceptance from vendors and developers alike (Jepsen, 2001). The WSDL is considered the interface of a web service and clients need to know the interface in order to know how to invoke the services. Like SOAP, the WSDL is also written using XML, making it parse-able. Typically WSDL files are stored in a UDDI where they can be found by the clients who want to subscribe to the services.

2.3. Service contracts

In Section 2.1, the WSDL was introduced and briefly described as the interface of a web service that contains the functional and non-functional characteristics and how information may be transferred between interacting services.

Terms that have been used to define a contract in the social context include:

a) A promise to perform

b) Terms and conditions for performance

c) Agreement with specific terms between two or more entities in which there is a promise to do something in return for a benefit

These descriptions of a contract are not too dissimilar to what a WSDL describes. The WSDL specifies a contract that a client accepts in order to invoke a service correctly and what the service provider promises to deliver upon correct invocation of the service offerings (Papazoglou, 2008). For any pair (provider | consumer) to interoperate successfully there must be a common understanding and agreement of what the provider offers and what the consumer can use. The WSDL document is considered the *de facto* standard for writing web service contracts (Erl *et al.*, 2008). The formal arrangement of the contents of a service, the price, the expected protocols for integration and quality aspects of a service are presented in the form of a contract. A contract is therefore defined as the service schema elements that are expected by the consumer and offered by the provider (Papazoglou, 2008a). The XML Schema expresses the shared language for defining the structure of documents and helps machines to carry out rules made by people. Web Service (WS)-policy specifies the behavioural expectations of a service and can be used to extend the contract provided by the WSDL and Schema (Erl *et al.*, 2008). Service Level Agreements (SLA), although outside the scope of this work, help establish the conditions and verifiable qualities of a service that a service provider should meet (Bianco *et al.*, 2008; Ruz and Baude, 2010). When a service consumer accepts the contract and can implement all parts of the contract to achieve all possible interactions with it, then the consumer is said to be compatible with the service.

2.3.1. Contract compatibility and Service Design

The notion of contracts gives a mechanism to define compatibility (Papazoglou, 2008b). It is necessary to maintain the same contract running for as long as possible in order to support existing consumers. It is also necessary to ensure that when services are developed and deployed they work properly (Bordeaux *et al.*, 2005), and that consumers can use the service on the basis of the contract

for that service. When are two contracts said to be compatible? Two contracts are compatible if they work together with no need for alterations to achieve interoperability. Compatibility has two variants, namely backward compatibility and forward compatibility, which need to be incorporated in web service design in order to sustain manageable web service evolution (Wilde, 2004). This calls for proper design of web service contracts. Proper evolution is not supposed to be destructive to current consumers and hence must remain backward compatible. When a contract evolves in such a way that there is no common feature maintained between subsequent versions, this is an altogether new service contract and not an evolved product.

2.3.2. Backward Compatibility

Already existing service consumers and service providers develop trust (Malik and Medjahed, 2010) when they become coupled by consumers binding to a web service. Backward compatibility implies that a newer web service version can interpret correctly and respond to the requests sent to it from older consumers. Developers therefore strive to maintain backward compatibility to avoid negative consumer impacts. This is the easier variant to implement (Wilde, 2004). That is to say, the new version of the contract has to continue to support consumers designed to operate with the old version of the contract which the consumer was using in a compatible manner, thereby maintaining compatibility and uninterrupted service in an evolved web service.

2.3.3. Forward Compatibility

Forward compatibility is difficult to incorporate in service design as there is no easy way to predict future changes that will need to be made (Karus, 2007). So contracts can be implemented with some degree of forward compatibility by adding optional elements. This allows for the existing

provider contract to seamlessly interoperate with a new version of a consumer with no modifications of the provider contract in any way.

2.4. Lehman's Laws of Evolution

Around 1980, M.M. Lehman grouped large programs into three types, giving rise to the SPE scheme (Lehman, 1980). This grouping was a classification of programs according to the relationship they have with the environment they run in. Studies reported in Lehman's work culminated in the Laws of software evolution as listed in Table 2.1. Having identified that web services are classified under E-type systems, it was also noted that not all the Laws of evolution will apply to web services, hence a selection of the Laws that were found to apply in this research work are as follows:

Law 1: Continuing Change –Once they have been published, web services are expected to undergo continuous changes to satisfy the consumers' needs. The changes are not limited to those coming from consumers but also from within management and the development teams in a bid to offer both improved quality and competition in a growing global market.

Laws 2 & 6: Increasing Complexity and Continuing growth: As long as the web service is improved and not replaced, it is evolving, which in turn makes the web service increasingly difficult to maintain and manage. The complexity of the evolved web service, is more than that of the original implementation and composition. In general, as web services are improved upon, they continue to grow and, more likely than not, the lines of code also increase.

Law 5: Conservation of Familiarity: As web services evolve for any given reason, there needs to be something common that is maintained which a consumer can identify with. During the lifecycle of a web service there may be updates, deletions and changes but once this gets to a point where

consumers cannot identify anything of the previous familiar web service, then it is no longer evolution but is considered as a replacement of an existing web service.

Table 2.1 presents a summary of the Laws of evolution from which the Laws that apply in the context of SOAP web service evolution were identified.

Table 2.1: Summarised Laws of software evolution (González-Barahona et al., 2014).

No.	**Brief Name**	**Law**
I 1974	Continuing Change	E-type systems must be continually adapted else they become progressively less satisfactory.
II 1974	Increasing Complexity	As an E-type system evolves its complexity increases, unless work is done to maintain or reduce it.
III 1974	Self-Regulation	E-type system evolution process is self-regulating with distribution of product and process measures close to normal.
IV 1980	Conservation of Organisational Stability (invariant work rate)	The average effective global activity rate in an evolving E-type system is invariant over product lifetime.
V 1980	Conservation of Familiarity	As an E-type system evolves all associated with it, developers, sales personnel, users, for example, must maintain mastery of its content and behaviour [leh80a] to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves.
VI 1980	Continuing Growth	The functional content of E-type systems must be continually increased to maintain user satisfaction over their lifetime.

VII 1996	Declining Quality	The quality of E-type systems will decline unless they are rigorously maintained and adapted to operational environment changes.
VIII 1996	Feedback System (first stated 1974,formalised as law 1996)	E-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

2.5. Literature Review

2.5.1. SOA governance (Service Change-oriented Lifecycle and Evolution)

SOA governance has been proposed as an approach to manage the evolution process of web services, where Information Technology (IT) governance refers to management and control mechanisms to ensure that standards, procedures and Laws are followed (Witte, 2013; Brown *et al.*, 2006; Schepers *et al.*, 2008). SOA governance extends IT governance and focuses on the life-cycle of services, metadata and composite applications in SOA (Brown *et al.*, 2006). The SOA governance process is aimed at managing the SOA lifecycle and should be applied through the four stages of modelling, assembling, deploying and managing a SOA. IBM's SOA governance and management method approach is based on the SOA governance process, seeking to engage customers in identifying reusable IT governance elements to build a new model (Brown *et al.*, 2006). The IBM approach seeks to establish an iterative governance method that a customer will follow to verify whether services and processes meet the objectives for which they are implemented. Schepers *et al.,* (2008) also advocate that SOA governance is not just a process but should be an ongoing alignment of strategic goals and the use of gained experience. With respect

to web service evolution, their approach focusses on controlling the service lifecycle through the use of service registries imposing validations on a service before it is published to avoid the cropping up of rogue services which cannot be governed (Schepers *et al.*, 2008). Although they acknowledge that there is complexity in the change management process in SOC and that consumers would need to be allowed time to switch to a new version of a service, there is no mention of how the customers will be informed and how their provider will be sure that no one is still using the older service versions. This might mean the older services still being maintained longer than the time Schepers *et al.,* (2008) called "temporary", in order to avoid service disruptions to the unknown consumers who are still using the older service versions. Other authors view SOA governance as the processes, policies, structures and practices that are in an IT governance program which, if implemented correctly, will enable an organisation to deliver reliable and high quality services (Afshar *et al.*, 2007; Bernhardt and Seese, 2009; Witte, 2013). SOA governance is identified as important in delivering an effective SOA, but does not directly address the issues in evolution of web services in SOC such as how exactly a web service is to be evolved and versioned while maintaining consistent service to existing consumers.

In complex Service Based Systems (SBSs), the services will evolve to meet changing user requirements and execution environments and what may hinder this is poor implementation or evolution. Evolving to meet the need for changing requirements may result in poorly implemented solutions, referred to as antipatterns. There are several solutions that have been proposed relative to some identified antipatterns by authors at enterprise level (Krai and Zemlicka, 2007). *Multi service* and *Tiny service* have been identified as common antipatterns and found to be the main cause of SOA failures. Moha *et al.*, (2012) proposed an approach with framework support for detecting these antipatterns. They reported that their solution had a precision of more than 90% in

detecting SOA antipatterns (Moha *et al.*, 2012). Their work ultimately contributes to the ease of maintenance and evolution of service based systems by assessing the design and quality of service metrics. Their approach looks at possible ways that SOA may fail, but does not present a solution to maintaining web services consistently while they evolve.

When web services are evolved by the service provider, they have an impact on the consumers of that service. Some customers may be forced to upgrade their consumers, some may discontinue completely and others may have unforeseen business impacts; it is therefore prudent to investigate the impact of the changes to determine how best to maintain web services with minimal or no disruption to consumers. Treiber *et al.,* (2008) investigated the changes of web services focusing on a single web service as the atomic building block for a composite business process fulfilling a web service. They argue that evolution of web services needs to be looked into from a holistic point of view as opposed to the focus of some approaches such as WSDL and WSDL-S (Akkiraju *et al.*, 2005) which single out interface related issues. Treiber *et al.,* (2008) identify the stakeholders that influence modifications on web services and describe the modifications in interface, implementation, requirements and quality-of-service for web services. This work does not directly address the evolution itself but sets a foundation for the process of web service evolution and identifies the changes that can occur in a web service as a whole.

Papazoglou, (2008b) also presents a theoretical approach to identifying and handling shallow changes, and a change oriented service life cycle methodology in SOA for deep changes. Web services can be offered to a specific set of customers where the provider knows the consumers who are using the service. Thus when service changes are effected they are localised to affecting the specific clients to that service and Papazoglou (2008b) refers to these as shallow changes. He reports that shallow changes can be handled through service substitutability, compliance and

service compatibility between versions. However, once services go beyond a point where they can have compatible change other mechanisms have to be employed that can ensure the continued support of old consumers while new service versions are introduced. Web services can also be published publicly, where in a typical SOA implementation consumers find and bind to that web service without any formal means of contacting the provider such that the provider is unaware of the actual customers using the service. This brings about complications in that the provider may not know the consequences of making a service change during evolution. Papazoglou (2008b) classified these changes as deep changes which can disrupt the whole business chain. In order to resolve or minimise the effects of deep changes, he introduced the change-oriented service lifecycle methodology to govern the changes as they occur. His work provides the theory which can be employed in practice to implement a model that may effectively manage the evolution process of a web service.

Espinha *et al.,* (2015) acknowledge that there are no specific industry standards being followed by organisations for evolving web services and that the rate at which providers change their service versions and the support thereof, is not uniform. Thus Espinha *et al.,* (2015) made the following recommendations for service providers to ease the burden of maintenance on client developers:

- Services should not be changed too often. (For example, Facebook was found to push breaking changes on a monthly basis)
- Old service versions should not stay in service too long as client developers tend to relax and not bother to apply the changes
- Use the technique of blackout testing such as the one employed by Twitter
- Examples of how to interact with the new versions of the web services need to be provided by the providers

The afore mentioned recommendations, if followed, provide value to industry and client developers, there still remain challenges such as, competition amongst providers which will necessitate for the continual improvement of existing services thus dictating the rate at which service versions will inevitably be introduced. Another challenge may be that there will be a need to maintain the service for longer periods as was the case where version 2 of Google Maps which had to be maintained for more than 3 years as opposed to the initial 1 year period for change that had been established by Google (Espinha *et al.*, 2015). This calls for a solution that is versatile and can accommodate both the provider's need to change and provide support for client developers who do not want to be forced to upgrade their applications but wish to do it in their own time and out of their own need.

2.5.2. Service Evolution Frameworks

Frameworks are the foundations or blueprints upon which predictions about the relationships between variables are based, while methods are the procedures that can be followed in implementing a research solution to a problem. Mosser and Blay-Fornarino,(2013) proposed a general framework they called the Activity meta-moDel suppOrting oRchestration Evolution (ADORE) framework to support process design and application of evolution on large processes, and detecting interference between the components. ADORE focuses more on the evolution of web services from a compositional perspective but does not drill down to the individual unit web services that are used in putting together the composite web service. Another theoretical framework was proposed for controlling the evolution of services that deal with changes caused by structural, behavioural and quality-of-service levels (Kajko-Mattsson *et al.*, 2007). In their work, Kajko-Mattsson *et al.*,(2007)identify the roles that are played by the teams in an organisation that are required for SOA-evolution. In their framework, they identify the SOA roles of evolution

as the responsibility of the SOA Project Manager and the Project members but do not explicitly highlight the actual aspects of service evolution that the manager or the members would be taking care of.

Lessons can be drawn from literature that may not necessarily be addressing services evolution concerns. For instance, combining Software Product Lines (SPL) and SOA has seen some techniques being applied to achieve high levels of reusability and, flexibility in software, enabling software configuration and customization (Dlamini *et al.*, 2013; Murugesupillai *et al.*, 2011). Variability evolution analysed in SPL can also be modified and extended to variability evolution in SOA. Other approaches reported in the literature addressing evolution include feature and variability management, where these concepts that are being extended from SPL to SOA. COVAMOF was developed as a framework for describing variability in SPL (Sinnema *et al.*, 2004), which Sun *et al.*,(2010)used to manage variability in web service-based systems. Another example of SPL concepts being extended to manage evolution through variability is the feature modelling technique. Feature modelling was also extended to managing and modelling web services in which feature diagrams are used to present the commonality and differences in features highlighting variability (Robak and Franczyk, 2003). We maintain however that, although evolution can be addressed by maintaining variability in web services from a composition perspective, this only addresses evolution of the composed web service and not the actual unit services that make up the whole. Neither does the work on variability management address service disruptions to consumers as each service evolves and how the effects can be minimised if not eradicated.

The Service Evolution Management Framework (SEMF) addresses issues to do with information provision and management while keeping track of web service changes as evolution occurs

(Treiber *et al.*, 2008). SEMF monitors evolution changes and analyses the dependencies among the changes implemented in a web service. Their work categorises some types of changes and sources of the changes, for example, the interface changes caused by the developers when they add/remove operations in the WSDL. Their work highlights SEMF as a useful web service information management tool but it is limited to monitoring web services for management and does not give information on how to implement and deal with the changes, or how to minimise the effects of evolution on consumers.

Frank *et al.*, (2008) describe their approach of hosting versioned web services which introduces a proxy to receive a request and redirect it to the correct service implementation. Their work shields the consumers from service disruption while new versions of the web service are implemented. The proxy acts as an intermediary between the consumers and the versioned service implementations, rerouting the service requests to the intended implementation. For each version of the web service that is implemented, a new proxy associated with that new version is implemented, and during what Frank *et al.*, (2008) describe as the transition states, multiple implementations of the service versions are maintained. While this work addresses the challenge of making version changes transparent to consumers, it does not take into account the economics of managing the multiple service versions that need to be maintained, such as what resources will be required to run the setup of multiple proxies increasing with each new version implementation and how much labour in terms of man-months the implementations will take. Neither does it take into account the challenges that are associated with maintaining the multiple proxies as the number of web service versions grow, given that the authors acknowledge there is no prescribed or explicit versioning strategy that is universally followed by providers.

The chain of adapters proposed by Kaminski *et al.,* (2006), promises to address the concern of backward compatibility in an evolving web service scenario. Perhaps the best way to describe the chain of adapters is by showing it – Figure 2.3 presents the solution proposed by Kaminski *et al.,* (2006). As apparent from Figure 2.3, the challenge existing as an essential constituent (inherent) of the chain of adapters approach should become evident – i.e., as the web service versions increase, the adapters increase. The distance from the interface v1 also increases along the chain, degrading the quality of service offered to the consumers still binding to that v1 interface. It is also important to highlight that there are maintenance challenges that come in to adjust and update each adapter along the way when a change that affects all consumer versions is to be implemented.

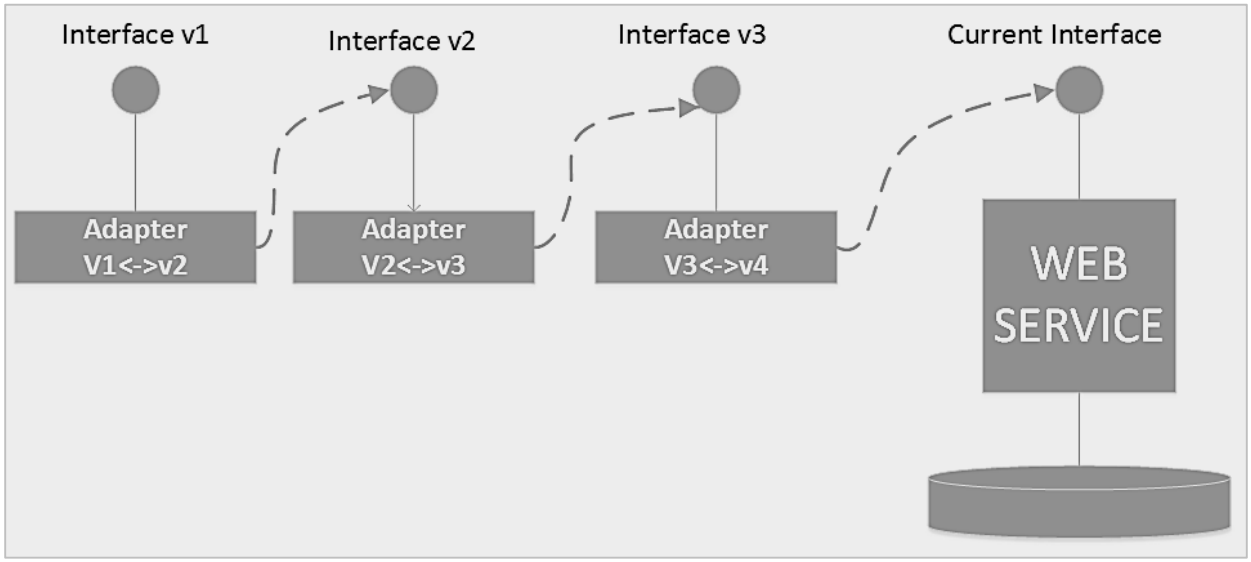

Figure 2.3 The chain of adapters technique (Kaminski et al., 2006)

The chain of adapters addresses some of the pertinent requirements in the maintenance of an evolving web service in the light of independently developed, unknown and unsupervised consumers. It maintains backward compatibility such that no old consumers relying on an older web service version are forced to upgrade in order to remain serviced, until the web service is formally withdrawn. The technique also employs a common data store across service versions in

an effort to make sure of the consistency of information served to consumers, this was felt to be an implementation specific detail that need not be a concern in web service evolution. Kaminski *et al.,* (2006) also implement automation in the creation of the adapters forming the chain, helping to ease administration of the adapters on the occasion of long chains of adapters. However, if the adapters are across a networked setup, this technique suffers a potential high performance degradation owing to the latencies not only in the network but also due to the operations in the adapters themselves. Thus a more compact solution might be more useful in achieving better results while managing the evolution of a web service effectively.

2.5.3. Service evolution methods and models

Zuo *et al.*, (2014a) proposed a model-driven method to manage web services by managing variability based on Model Driven Architectures (MDA). The service industry indeed needs tools to support the actual evolution of the services they provide. Few of the works propose development of support tools for SOA evolution (Zuo *et al.*, 2014a). Zuo *et al.* (2014a) presented a model in which they describe the stakeholders in a web service in SOC. In their model, they introduce a service broker as having the roles of maintaining the registry in which services are published and notifying the customers who are interested in the service, of the service's changes as they evolve. Consequently, customers are expected to upgrade their consumer instances upon receiving notifications. This model is expected to work well in the case where customers are known to the provider or where consumer agents are intelligent enough to continuously check the registry for service changes in what is classified as shallow changes (Papazoglou, 2008b), but in the case where customers are not known, the model will not be able to support the older consumers who rely on the older version. Customers are forced to upgrade their consumer implementations in the case of

this model, which may cause unforeseen and unbudgeted-for development expenses, and that would be a disservice to the customer.

Mingyan *et al.*, (2008) proposed what they called the Service-Oriented Dynamic Evolution Model (SOEM) to manage the dynamic evolution of web services. They argue that the concepts that had been put forward showed that SOA was better at evolution than the traditional development patterns as SOA added a service level uniting the business logic and the technology levels. Evolution where the system need to restart in order to accommodate an evolution update is referred to in their work as static evolution and hence they usher in SOEM. SOEM is described as dynamically evolving on the basis of a user request relying on a service bus to decide on the evolution of the web service. When the evolution is completed other users will then see the evolved service. SOEM does not, however, reveal how the evolution will be controlled since there is a need to have someone or something in place to make sure the evolution requests from users maintain the evolved product in a consistent manner. The evolution of the service also needs to be in such a manner that other users are not affected negatively, while it is mentioned in their work that, once a service is evolved, all other users see the new service. This poses a challenge to users who may not wish to upgrade and rely on the older state of the service.

Most solutions to web service evolution that researchers have provided contribute to varying aspects of the bigger picture which includes service providers, customers and infrastructure (Zuo *et al.*, 2014b). Zuo *et al.,* (2014b) proposed a change-centric holistic model for managing web service evolution which factors in the changes that have been made to a service, when the changes occur and how to apply the changes including performing client adaptation to the changes implemented. In their work they employ a similar technique to the chain of adapters, Kaminski *et al.*, (2006), and improves upon the chain by limiting the number of endpoints in the chain. This

work however assumes that they have control over the customer implementations, which may not be true in a SOA deployment.

Compatibility is key to the usability of any service. If a service is evolved from one version to the next, the newer version needs to remain compatible with the service customers otherwise service is lost (Vara *et al.*, 2012). Vara *et al.,* (2012) present a domain specific language toolkit that has the reasoning to determine the compatibility of subsequent service versions. They compare abstract service descriptions and conclude the compatibility on the basis of similar elements and relationships being in both versions. Primarily, the work presented by Vara *et al.,* (2012) reduces the work that has to be done parsing WSDL files to look for compatibility and provides a good foundation by which compatibility can be ascertained. However, their work focusses on non-breaking changes between services, whereas breaking changes also need to be addressed in an evolving web service implementation. In conclusion, their work gives a possible future research direction of using service contracts to achieve flexibility in evolving services resulting in compatible outcomes.

2.5.4. Contract Evolution

As web services are updated, and new functionality is implemented, there can be reduced quality in the service offerings. For instance, the size of the request message may increase, and that impacts the processing time for each request. Gorinsek *et al.*, (2003) discussed a methodology for managing the quality of service of components in an evolving system. Their methodology uses component contracts to determine the impacts of an update on the running implementation and on the resources available for the program. Using contracts provides the information on how components behave and their expectations to its neighbour components and the amount of resources the component requires. This helps in the area of dynamic systems adaptation and

evolution where a component can choose services from another component on the basis of understanding the component that can complete the task within the available resources and acceptable quality of service parameters. Though this work was not directly concerned with web service evolution, it extends the notion that evolution of a software service can be achieved and managed through the contract and Andrikopoulos *et al.,* (2012) formalised the use of contracts in determining compatibility of web services.

Contracts promote the separation of implementation and description of the web service, thereby enhancing the loose coupling feature in a SOA implementation. Similarly in legacy systems, contracts enable systems to be designed and developed in a compositional way which ultimately improves flexibility in an evolving system (Andrade and Luiz, 2000). In their work, Andrade and Luiz focus on how component-based systems can leverage the notion of a contract in achieving evolution of the system by replacement and addition of contracts without necessarily changing or affecting the whole system. The contract, in their view, consists of classes and the roles they play in the system and the effect description of the behaviour that the other components should expect. Though it is a very useful notion to use contracts in managing compositional evolution while they (Andrade and Luiz) claim to have increased levels of flexibility in the development process on component-based systems, their focus is on evolution by contract of a composed system. There still remains a need to focus on managing the evolution of the individual components themselves which the contracts describe.

Evolution of web services has also seen attempts at evolution management through versioning. There are several versioning strategies that have been described in the literature, each with its advantages and disadvantages (Erl *et al.*, 2008), for example:

- Strict versioning strategy, which disregards backward and forward compatibility but gives full control over contract evolution. Any change in the contract would imply a new version has to be implemented
- Flexible versioning strategy, which considers maintaining backward compatibility supporting existing consumers, but the changes are irreversible as a reversal would bring about incompatibilities
- Loose versioning strategy, which has policy assertions that are optional and wildcards in the contract design allowing for an expansion in the acceptable messages and data content. Though this strategy caters for both backward and forward compatibility, it presents challenges with message validation since some features are optional and leave the validation to be handled by the implementation of a service

Contract versioning is not sufficient on its own to address the challenges of web service evolution. It leaves challenges such as how version changes are to be communicated to consumers as these consumers are not intelligent and are statically built and bound to a particular existing version. Another challenge lies in deciding whether to maintain an older version and for how long in order to maintain older consumers, then also determining the impact on consumers if the older version support is dropped altogether.

2.6. Summary

Web services are experiencing an increasing infusion into the business environment. Web services are taking over from legacy (in-house) systems running single business tasks to a service-oriented environment where sub-tasks are subcontracted to services offered across the Internet from other service providers. Composed services complete a business process by cooperating with other

providers across the world. This has changed the ways in which software is managed in the light of the fact that not all of the service is offered by the same provider, hence there is a need to ensure interoperability even after a services are changed or updated. These service changes may affect the consumers relying on that service and disrupt business processes, resulting in possible huge losses.

In general, all the studies reviewed in this Chapter are not lacking in insight, contributing some technology, method, framework or even tools, towards the management of evolving web services in SOA. SOAP web services provide a convenient means for describing what the service offers and how it can be used through the WSDL file which is the essential component of the service contract. Consequently, the contract can be used to effectively develop web services and extend the loose coupling in web services and legacy systems. Service versioning and contract versioning help in managing the changing services but that still leaves the challenge of communicating the changes to consumers who are unknown and independently developed.

Other approaches that have been proposed concentrate on the evolution of a composed system as a whole, leaving the evolution of the web service, as a unit, to the provider. The challenge, again, that remains in such approaches is that they do not address directly the effects of the unit evolutions on the composed web service as a whole. On the other, hand there are authors who have looked into unit evolution and proposed adapter approaches and multiple version hosting. These approaches go a long way to maintaining uninterrupted services and maintaining backward compatibility with the old consumers, yet they overlook the challenges and expenses incurred in a real-world implementation of the suggested solutions.

It is in this light that the solution of a contracts-based proxy model for web service evolution management can bridge the gap between the theories of contracts in evolution of services, the maintenance of an evolving web service as a unit and the cost implications in terms of time, labour

and resources. In the following chapter, we describe a possible scenario in which the solution can be applied.

CHAPTER THREE

3. RUNNING SCENARIO

This chapter introduces the running scenario which is used from this point on to show the applicability of this work in real life. The goal of this work, however, is not to reinvent the wheel in web service development but to manage the evolving web services, hence the running scenario considered in this study is based on an existing SOA hotel reservation application that was developed as a common platform case study by other scholars at the Department of Computer Science, University of West Florida.

3.1. Introduction

It has been noted that software maintenance research does not have a standard case study that can be used to develop research ideas and compare and benchmark solutions (Espinha *et al.*, 2012). Isolated research, though not entirely bad, reduces the rate of advancements in research. It is in the light of this that this research work does not re-engineer web services used herein, but builds upon an existing system and implements a versioned StockQuote services upon which the tests are based. It is also well established in different fields that having such a standard case study system brings many benefits in that it helps determine which approaches work best for specific problems. For comparative results, there is a need for the use of a common case development and test platform. A web services based hotel reservation system was developed and made freely available as a teaching and research platform for Services Oriented Computing by the Computer Science Department of the University of West Florida (Wilde *et al.*, 2012). The web services based hotel reservation system is an integration of mainly two applications; the hotel reservation and Currency Exchange applications which were both implemented using PHP. The hotel reservation application

requires the Currency Exchange application, which provides the quotations for the exchange rates on currencies. In the implementation of the web services based hotel reservation application for this work, Currency Exchange implemented using PHP was replaced with the StockQuote web service implemented using java. The StockQuote web service offers the same service of quoting forex exchange rates, and more options were added as the StockQuote web service was upgraded. It is upon the StockQuote web service application that the evolution of web services was demonstrated and the model applied and evaluated.

3.2. Description of the Scenario

Today's software development is largely composed and integrated through web Application Programming Interfaces (APIs), with large API providers like Google, Amazon, Twitter and Facebook allowing for programmers to interface with their systems through APIs which are essentially web services. A study by Espinha *et al.,* (2015) revealed that client developers face a number of problems when web service providers evolve their services since there is no standard policy governing the evolution thereof (Espinha *et al.*, 2015). Twitter and Google were found to use versioning while providing up to 2 years for client developers to migrate to a new service version. Facebook, on the other hand, was found to push out breaking changes every 3 months irrespective of who is affected, leaving client providers to face the consequences of these changes on their own. Client developers had this to say: *"Facebook continually alters stuff thus rapidly outdating my apps" and "[…] [the biggest headache] is the never ending changes to the API"* (Espinha *et al.*, 2015). Though Google gives up to 2 years for clients to upgrade, *"if you have other projects, if you have to make money on other projects, even in the two years it is difficult to find time to implement [the changes]"* (Espinha *et al.,* 2015).

Picture a hotel reservation system. It sounds simple and appears to be just another software until you look under the hood. The reservation system in this scenario is composed of multiple services provided by various organisations all to orchestrate a business process of booking a room(s) for a customer(s).

3.3. The Hotel Booking Service

The hotel reservations system is a SOA application which allows users to reserve hotel rooms in any country from any of the hotels registered in the system. Using the reservations system, a customer is expected to provide information required by the specific hotel. These details will include full name, country of origin, credit card details, currency in which the client wishes to pay and other relevant details such as contact details. When a customer chooses a destination country and region, a list of available hotels, rooms and pricing is provided for them to choose from. If the customer chooses a room and provides the duration of stay, the system invokes the StockQuote automatically to provide the best exchange rate on offer from the banks that conduct currency exchange. The system then computes the total to be paid in the local currency to the customer and displays it to the customer for confirmation of transaction. The customer is given a limited amount of time in which to approve the quoted pricing and book the room, as exchange rates fluctuate quite rapidly in a trade period.

Figure 3.1 shows the SOA based hotel reservation application activity diagram in which the customer chooses the location and room to book. In the solution implementation the assumptions made are that:

- Hotels in various destinations are already registered as members in the reservation system

- Forex houses and banks that trade-in forex, are also registered in the system and their in-house services are queried for quotations on the specific currencies in which the payments to hotels will need to be made
- The StockQuote service is not limited to stock on trading floors but checks the registered banks and forex houses for their offers in a particular currency and returns the best rate on offer
- Once the customer confirms a booking the customer's bank purchases the forex and makes the necessary payments to the selected hotels

Figure 3.1 is a model for the logic that the implementation follows in booking a room. To use the system, the customer fills in the online forms with their personal details and credit information. The system validates the form input and displays a page for the customer to choose the location, the duration of stay and the currency they wish to settle the payments in. Once the customer enters this information, the systems invokes the StockQuote service and the credit worthiness check service. The StockQuote service queries the registered forex banks and companies for the exchange rate of the destination country and chooses the best rate on offer. The hotel reservation system then receives the currency rate from the StockQuote service, displays the hotel rooms available and computes the pricing in the currency chosen by the customer. If the customer chooses a room, they can confirm the reservation of their room and the system initiates with the banks, the purchasing of forex and the settlement of the payments with the chosen hotel. Had the credit card been declined or invalid, the transaction would be cancelled and the customer would be informed, otherwise the hotel generates a receipt and issues it to the customer.

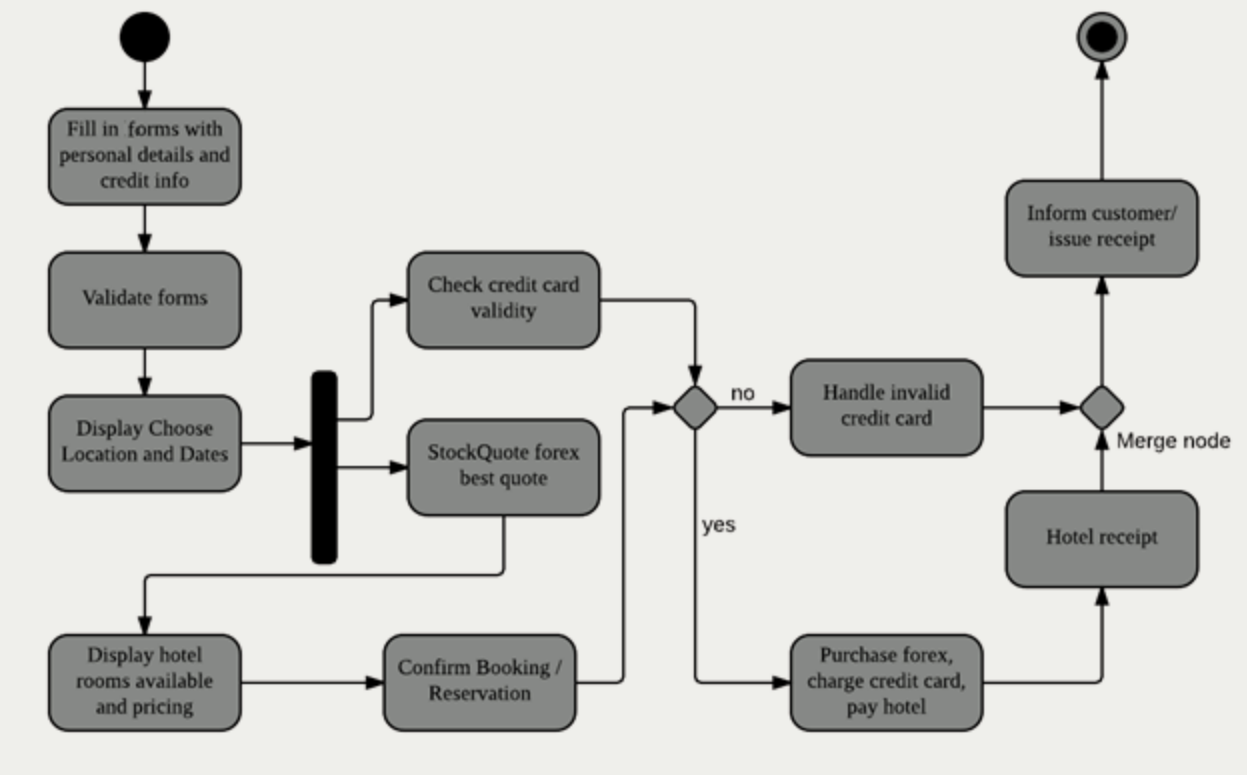

Figure 3.1: Hotel Booking Service activity diagram

The StockQuote service was developed independently of all the other services listed in this composite SOA hotel reservation system. The hotel reservation system is a consumer of the StockQuote web service as it relies on the StockQuote for currency exchange rates in order to quote and bill the persons making reservations. The StockQuote service is not only useful for providing current exchange rates for the hotel reservation application but can also service other consumers with slightly different needs such as stock history and real-time quotes as opposed to static quotes. The possibilities of the service being modified to address such requirements from other users outside of the hotel reservation application, brings about the challenges of evolving the StockQuote web service consistently while maintaining service to the hotel reservation application as an already existing consumer.

3.4. The Stock Service

The StockQuote service is queried to return the current exchange rate which each bank is offering. However, seeing as this is a service used by a variety of consumers who have varying needs and for all intents and purposes varying uses for the requested information, they have different expectations of the service offering. Investors increasingly assess stocks to decide when to trade and when to invest. This is being done online via limited profile mobile devices, smartphones, computers and tablets more than print media as the value of stocks change several times in a day as trade progresses. Print media offers stock quotes that are correct as at the end of the previous day close of trade, while some websites may offer static delayed stock quotes and last trades prices, while there is also the possible extension of offering real-time stock quotes to subscribers. The variations can be viewed as changes that can be effected on the StockQuote services and this work investigates the evolution of web services in this context.

3.5. Evolution Scenarios

To model the evolution of the StockQuote service, the service in this work was evolved into at least three versions for demonstration purposes, which were used for investigations into web service evolution. StockQuote version 1.0 was the first implementation and is a version which returns the value of the currency/price to the clients after receiving the stock symbol as the invoking parameter. Figure 3.2 shows a sample client request message invoking the web service with the stockSymbol “zar” and receiving the static price of “110.25” as the value for the day. A service client may request real-time quotes instead of the static standard ones as investors may need to make business critical decisions based on current information. This prompted the evolution of StockQuote service version 1.0 to version 1.1 which was enhanced so that a client can send an additional parameter indicating that they need real-time quotes.

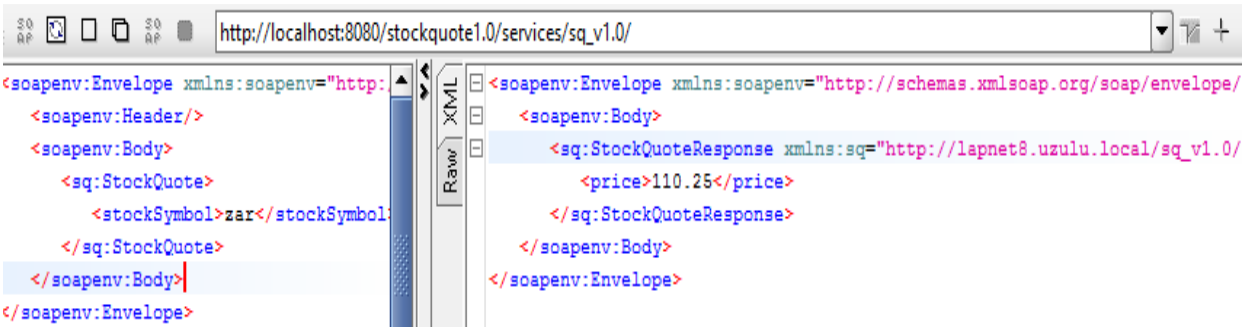

Figure 3.2: Client request and response for StockQuote version 1.0

As shown in Figure 3.3, the service at version 1.1 returns the minimum and maximum value of that stock symbol on the previous day after receiving the optional realTime value of "True". Without the optional value, the web service at version 1.1 would treat the request as a version 1.0 request, responding with only the price information. The version 1.1 has to process a slightly bigger payload and send back a larger payload, the result of which is discussed in Chapter 6, Section 6.3 of this work.

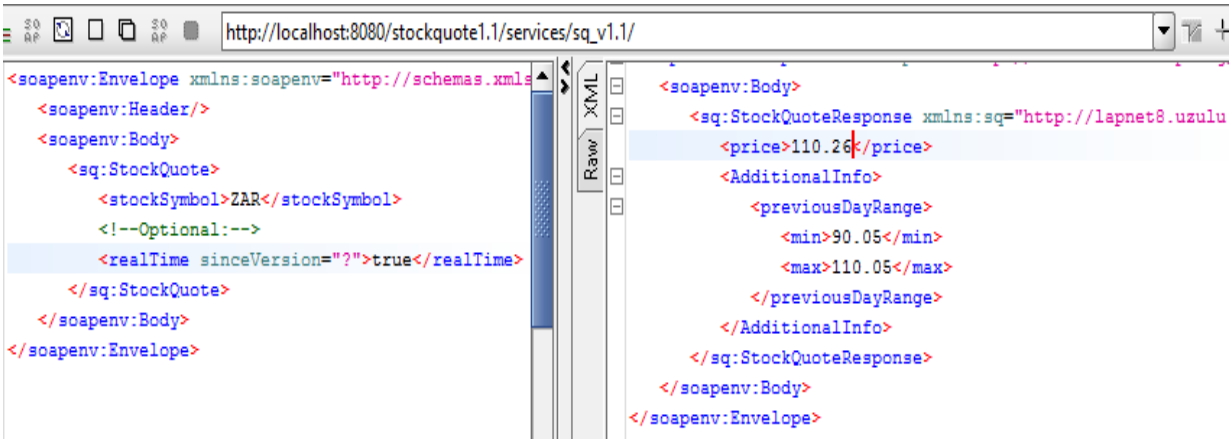

Figure 3.3: Client request and response for StockQuote version 1.1(Chiponga et al., 2014a)

There is a further enhancement in the next evolution step of the StockQuote service where in addition to the minimum and maximum stock values, Figure 3.4 shows service version 1.2 also having an optional parameter that returns earnings related information to the service clients. This

additional information shows an increased payload in the web service response, inevitably further degrading the quality of service.

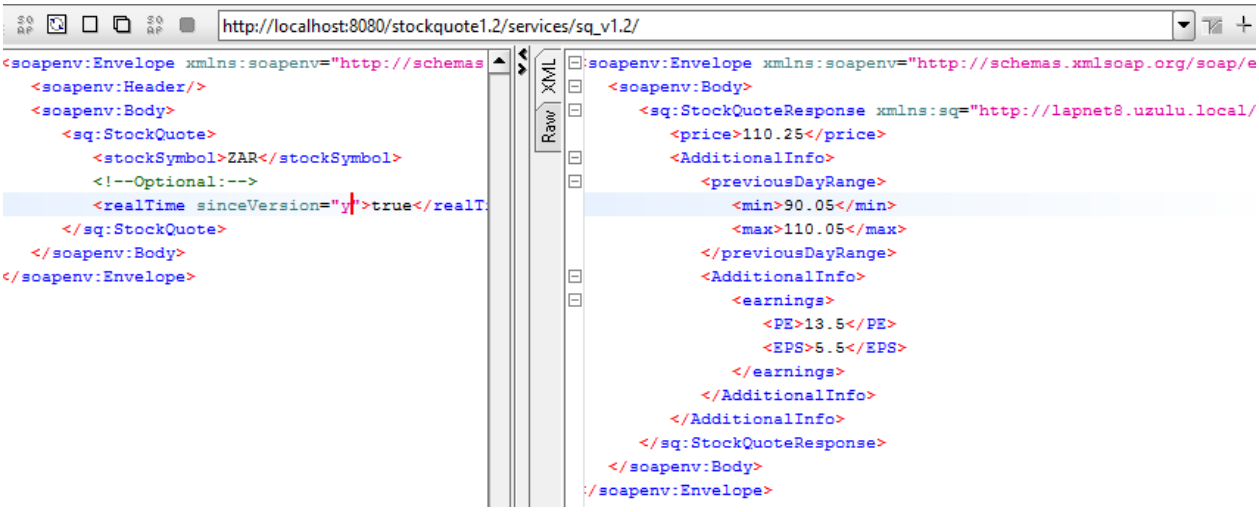

Figure 3.4: Client request and response for StockQuote version 1.2

There is a contractual agreement between the service provider and the service customers that the service upgrades are only available to the service clients that have been upgraded or implemented using the contracts (interfaces descriptions) matching the upgraded service. The old clients will continue to function in the normal manner even if not upgraded.

The hotel reservation system used in this scenario can be likened to Airbnb, which is a large community online marketplace connecting travellers with accommodation facilities in over 190 countries and 34 000 cities and has a listing worldwide of over 1,5billion (“Airbnb,” n.d.). Catering for large volumes of people such as the ones implied by the figures stated by Airbnb, would need careful and flawless management for the system as it evolves. Imagine the possible ramifications that could result from failed system evolution, and the companies, countries and people that could be affected.

Though the web service evolution as described in this chapter offers economic value to the service provider by allowing for a larger base of different subscribers and to the customers by offering updated information and options, if a service version is discontinued by the provider there are service disruptions to the consumers relying on them and losses that may be incurred by the consumers, hence the evolution process of a web service needs to be carefully managed. Chapter 4 discusses the design of the model to manage the evolution of a web service.

3.6. Summary

The running scenario that will be referenced through the remainder of this work was described in this chapter. The scenario shows how developers of APIs (web service consumers), constantly face challenges when service providers upgrade their web services. The hotel reservation scenario described in this chapter consists of web services, one of which is the StockQuote service. The StockQuote service was upgraded, resulting in three service versions that still require support to be maintained for the sake of the consumers using these versions. The remainder of this work focusses on building a model that can be used as a solution to the challenges associated with the maintenance activities of these service versions. Chapter 4 describes the design and reasoning behind this solution.

CHAPTER FOUR

4. DESIGN OF THE CONTRACT-BASED MODEL

Chapter 3 described a scenario that presents some of the challenges that are experienced by developers maintaining their services and applications that rely on web services. A request to make payments in the hotel reservation application would fail if the underlying web services were removed or changed in an incompatible manner to the current implementation of the reservation application. This chapter presents the design of the solution that is implemented in this research.

Epistemologically, a research activity need to have a sound basis in theory and knowledge to ensure that rigorous research is done (Olivier, 2009). If a project follows a generally accepted perspective, pattern or model in a particular discipline, it is highly likely to be qualified as research.

Sometimes, common sense leads researchers to assume that no tests are needed, but this causes a bias towards what the researcher would think is obvious. Following a scientific methodology attempts to minimize the influence of the researchers' bias towards a particular outcome of the work. There is no single universal scientific method, but that the researcher has to tune the processes to the type of problem under study. There are two main types of research approaches to a problem, quantitative and qualitative research. Quantitative approach generates data that can be quantified while qualitative focuses on verbal data as opposed to numeric values. This research work falls mainly under the quantitative experimental research method, which involves standard practices of manipulating quantitative data that can be analysed and repeated or nullified. However, it also uses some elements of qualitative approaches. Hence, one can correctly classify it as a blended approach.

Research in computer science predominantly follows the following methodologies: Modelling, Theoretical and Experimental Computer Science, computer simulation and controlled experiments. This research work combines theoretical and experimental computer science, within which, the paradigm followed by this research is design science. Innovative design calls upon design science research, where the focus is on developing artifacts to solve problems in the real world. Design science has been used mostly in Computing Science and Engineering to design artifacts such as algorithms, process models and human/computer interfaces, with some leading research institutions like Stanford's Centre for Design Research, MIT's Media Lab and PARC in Xerox (Haynes and Carroll, 2007) employing the design science research methodology (Hevner *et al.*, 2004).

This chapter presents a model which is the design science artifact that will be used in minimising the effects of service change in response to the second research question. In this model, a record of at least two previous service contracts is maintained as proof of the concept that a service has evolved according to Lehman's Laws of evolution summarised in Table 2.1. Hence, through these older contracts conserving familiarity (law number 5) one can trace the evolution of the service over time.

The literature survey conducted in this research revealed that there is no standard practice in the industry that every web service provider is legally required to follow. As a result, new versions of services come and go as fast as the provider dictates, irrespective of the clients they serve. Organisations such as Google can afford to maintain older versions of a service running for longer periods extending for up to 3 years or more (Espinha *et al.*, 2015), but this will not hold true for all application providers. However, with the aid of a suitably designed proxy, multiple versions can be maintained by any organisation for longer periods without needing extra resources.

For illustration purposes, the model is presented as maintaining the current contract version, contract (Vn+1), and the previous contract (Vn). The actual number of service contracts maintained can be more than the two, used for illustration, but limited to the organisational polices of the organisation in question. This chapter also sets out to achieve the third research objective, namely to develop a service message transformation proxy model relying on contracts to handle the incoming and outgoing SOAP messages between the client and the provider.

4.1. Conceptualisation

Conceptualisation is meant to give a mental picture of the model that was designed. Conceptualisation can be viewed as the concepts defining the relationships in the domain of web service evolution. These concepts set the limits and context in which web service evolution can be described.

Figure 4.1 shows the flow of logic within the contracts-based model. The provider, registry and customer are the actors that initiate an active part in this model while the consumer, proxy and web service can be considered as the objects or classes that respond to messages through invocation of an operation. A service provider builds a web service and publishes the contract of that web service into a registry. Once successfully published the registry keeps a record that the web service exists and is available for discovery. A customer wishing to use the service finds the web service in the registry and builds a requestor agent called a consumer. The consumer invokes the service by sending a service request message which is received by the proxy. The proxy collects the contracts of the web service versions it is supporting, checks if there is a need to transform the message or not and performs the necessary action to the request message. The proxy forward the request to the actual web service for processing. The web service responds with the reply to the invocation

by the proxy, and the proxy performs the necessary actions on the response, if needed, before sending the response message to the consumer.

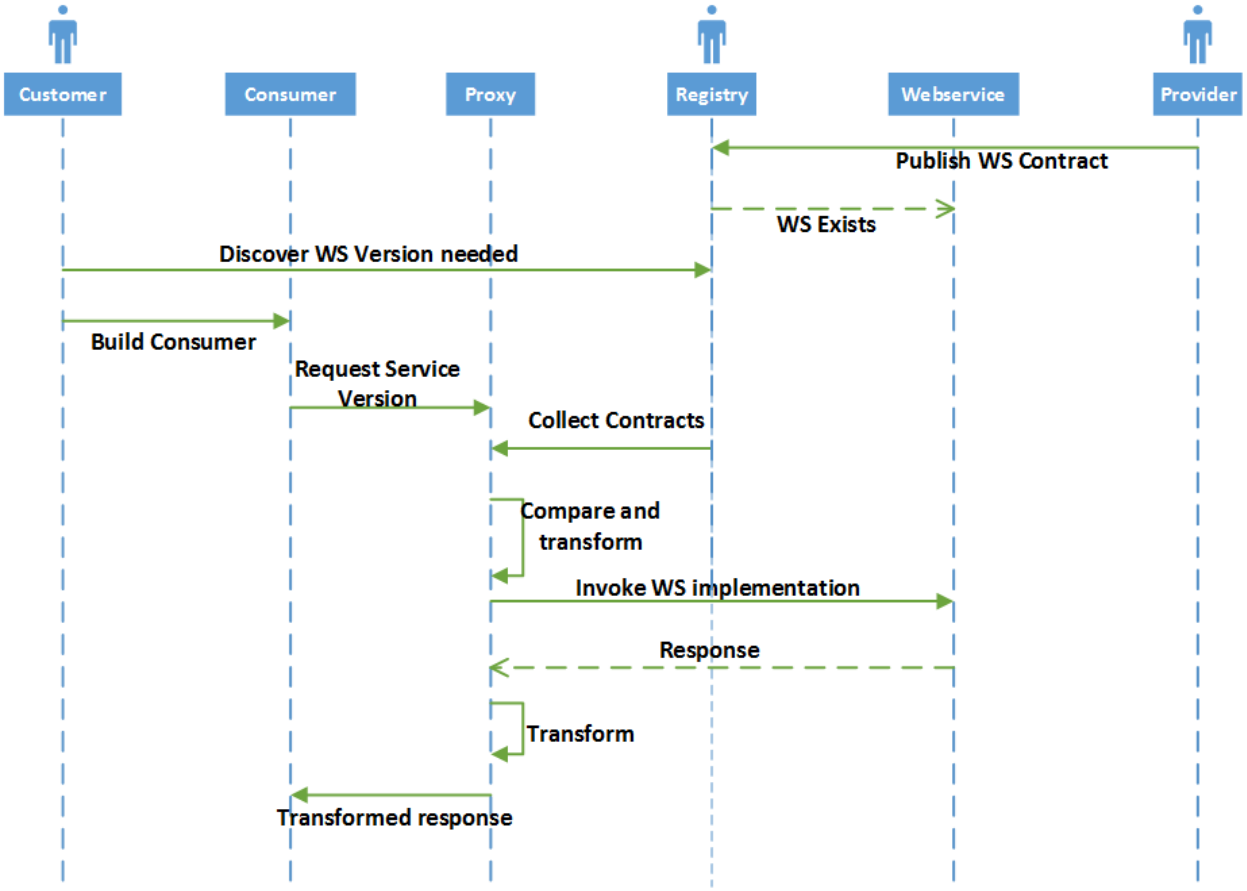

Figure 4.1: Sequence diagram for the contracts-based proxy model

The model is expected to receive incoming requests, understand which service version is being requested, perform the necessary operations on the incoming messages and deliver the correctly formatted message to the running web service. To achieve these capabilities, some thought went into the design of the functionality of the model, hence a set of design criterion was drawn up.

4.2. Design criteria

The contracts-based model was developed to support the seamless evolution of a web service from one version through to the next in its lifespan while also supporting active service to consumers. In particular, the design criteria that were laid out for the model were as follows:

- The model must be able to perform routing of messages between the web service endpoints and consumers. This can be achieved using content-based-routing, where a message is routed using the contents of that message. The routing includes choosing the correct path for a targeted transformation of a SOAP message and forwarding the message to the correct endpoint of the running web service
- The model must be able to match the request messages to the contracts determining which transformation paths the message will need to be routed through. The key feature of the model is the ability to match the SOAP request from the consumer to the running web service version and back to the expected SOAP response for the specific consumer that has invoked the service
- The model must be able to mediate messages between service requestors and service providers. For example, if a request message matches the most recent/ running version of the web service, no transformation will need to be performed. The request will be entirely compatible with the running web service so it will be forwarded as it is. If the message is matching an older service version, then the model is expected to perform the necessary transformation and deliver the correctly formatted response to the correct calling consumer
- Transparency is the key to the model's utility. The changes to a web service using the model should neither be visible nor disruptive to existing consumers. It must appear to the

consumer as if the old service version still exists while the actual service has been replaced. The only expected downtime may perhaps be during scheduled maintenance

- Availability is always a key requirement for any system in use and the model needs to exhibit maximum availability of the supported web service versions. The model needs to maintain an uptime of 99.99% where possible, leaving room for scheduled maintenance per month (Kern, 2003) and also needs to be responsive timeously for it to be considered "available"
- The model must maintain records of the previous versions so that the evolution from one stage to the next is traceable (Zuo *et al.*, 2014b). It is considered to be evolution if one can trace back the history of a web service, and if possible, all the way back to the original version

4.3. Best Practices in Software development

4.3.1. Software Configuration

Software Configuration Management (SCM) is needed in controlling the evolution of software systems. Systems are composed of parts that have varying names such as interfaces, modules, components and subroutines. These components are identified using versioning and as such have some version numbering associated with them. The different parts have interfaces that are compatible with some interface versions and not others. Some best practices key to a successful system development/evolution in SCM can be summarised as:

- Identify and store artifacts in repositories: there is a need to identify the artifacts that would need to be versioned. Appropriate version control schemes need to be used to version them and store the artifacts in secure, scalable and accessible repositories

- Control and audit changes to artifacts: the updates to an artifact need to be controlled and any changes made need to be tracked. This may be in the form of documentation so that any changes that introduce errors into a system due to changes in an artifact can be easily rolled back
- Record and track requests for change: changes made to any system involve a request for the change being made. The change may emanate from management, platform upgrades or users of the system and good change management will enable good project management and decision making

Keeping a record of the changes that have been made is one of the best practices in SCM and this enables traceability of the evolution steps that have been applied to the web service as it evolves. Records of the existence of a web service can be kept in a registry that acts as a repository for web service artifacts. The repository maintains a copy of each published contract. Each contract (WSDL) has enough detail to identify the web service version and the parameters expected, giving enough information to track all the changes that have been implemented across versions. The model developed in this work was designed with the Lehman's Laws of evolution and SCM in mind, where a record of the previous version(s) of the web service has to be maintained for traceability. In keeping with the law of conservation of familiarity and best practices in SCM the model also maintains that there has to be some common element(s) in each subsequent version of the web service for it to be called evolution, else the new web service version without anything in relation to the previous, becomes a totally new web service. In order to achieve controlled evolution the Top-Down Approach (also known as the Design by Contract approach) was used to develop the web services.

4.3.2. Contract Design

Design by Contract (DbC) is a design methodology that ensures software correctness by using preconditions and postconditions to document (or programmatically assert) the change in state caused by a piece of a program. DbC is a trademarked term of Bertrand Meyer (Crocker, 2004), which he implemented in his Eiffel Language as assertions, as the mechanism for expressing the pre and post conditions of any subroutine handling a subtask. Parallels can be drawn between web service composition and Object Oriented Programming (OOP) in that software components used in OOP need to be built on the basis of carefully designed contracts. In SOC there is a binding between the web service consumer and the web service provider which is specified by the pre and post conditions in the WSDL, which is considered the contract in the case of SOAP web services. Through the contract published by a service provider in the UDDI, each consumer binding to a service expects some or no result after invoking a web service and on the basis of the same contract, the web service provider expects to receive some parameter or more than one in order to deliver the promised response in a timely fashion (as stated in the assertions).

The WSDL and XML Schema Definition (XSD) are technical components making up a web service contract. Web services can be designed following the Bottom-Up-Approach in which the development of the web service starts with the implementation (e.g. Java methods), and the WSDL file is generated from the implementation. The Top-Down-Approach, also referred to as the *Design by Contract,* focusses on setting up the contract first, WSDL, and all relevant schemas followed by the implementation (Hollunder *et al.*, 2012).

Following DbC presents advantages such as defining the XSDs separately and allowing reuse of the same XSD file in other relevant scenarios, and there is looser coupling between contract and implementation as the contract is not dependent on the implementation. Loose coupling allows

versioning control and performance is enhanced as one has more control over what is sent over the wire, than when java is converted automatically into XML. DbC allows us to design custom contracts that point to the same endpoint, hence there is no need for the proxy to advertise multiple endpoints for each web service version and in essence no need for upgrading customers to concern themselves with any change in the physical interface.

4.4. The model setup

4.4.1. The ideal provider and consumer setup

Typically, web services are deployed on different machines on the Internet and are exposed to the rest of the world by the service providers through the web service interfaces. In Figure 4.2, a service provider exposes a service's functionality using the service interface to which consumers direct all their service requests. A service consumer invokes the web service by sending a service request to the web service interface and receives a service response from the service provider via that interface. Communication between the service provider and consumer may be in the form of simple messages or parameters passed in the Uniform Resource Identifier. In the case of SOAP web services, the service request and service response are in the form of SOAP messages.

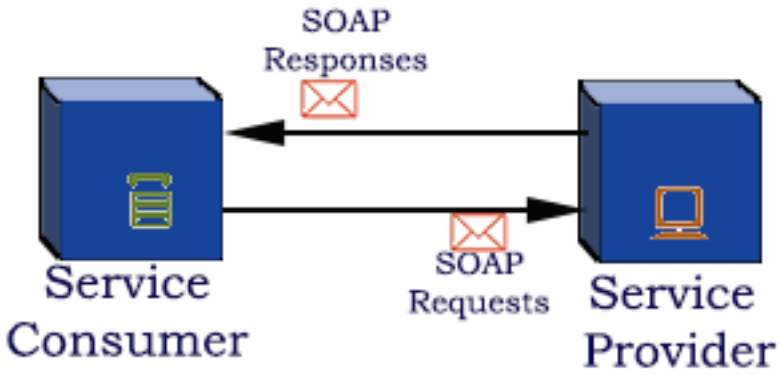

Figure 4.2: The ideal relation between service provider and consumer in SOA

4.4.2. The Realistic Provider and Consumer setup

A service can be both a service provider and a service consumer at the same time as it can consume other services. As illustrated in Figure 4.3, the consumer and provider communicate using a communication medium in a local area network for internal services or the Internet for external services. This communication medium is called a service bus. The service busses offer real value in scenarios where there are a few integration points or at least three applications that need to be integrated. Integration is a process commonly known as service composition, where a business process is accomplished by combining atomic services to complete subtasks of the main business process. Service busses exhibit capabilities such as message routing, transaction management, message security, transportation and transformations from one message format to another, which web services can leverage upon. Service busses are also well suited to scenarios where loose coupling, scalability and robustness are required. The service bus is an abstract pattern employed in the transfer of data between multiple systems following some common policy to send and receive messages or to route messages between systems. Web services are designed for machine to machine interaction. This presents the advantage that any other platform can be used to consume the service and present the information offered by the service in any format. Web service interface provides a standard and open platform understood by the service consumers and the service providers without needing to restrict to a particular programming language or programming environment and hardware.

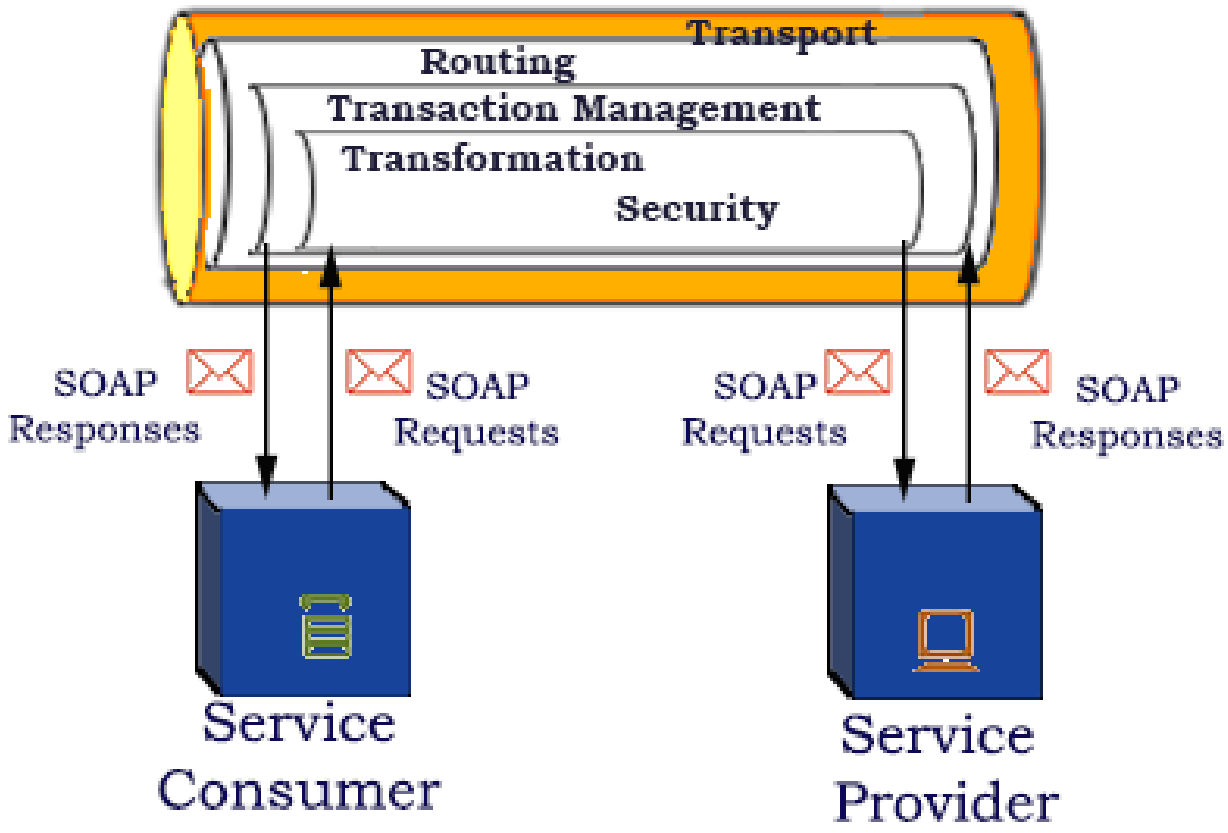

Figure 4.3: The realistic typical service consumer-service provider request.

With the realistic setup in mind Figure 4.4 depicts the resultant contracts-based proxy model for managing the evolution of web services. Figure 4.4 shows the service provider **publish**ing the versioned contracts of the service to the service registry. The consumers **discover/find** the service and **bind** to it and then invoke the web service through the interface defined in the contract. The SOAP requests sent by the consumers are received by the proxy, which performs that necessary transformations as described later in this chapter. The proxy forward the transformed request to the web service hosted by the provider. The proxy receives the response from the web service and routes the response to the consumer that invoked the service.

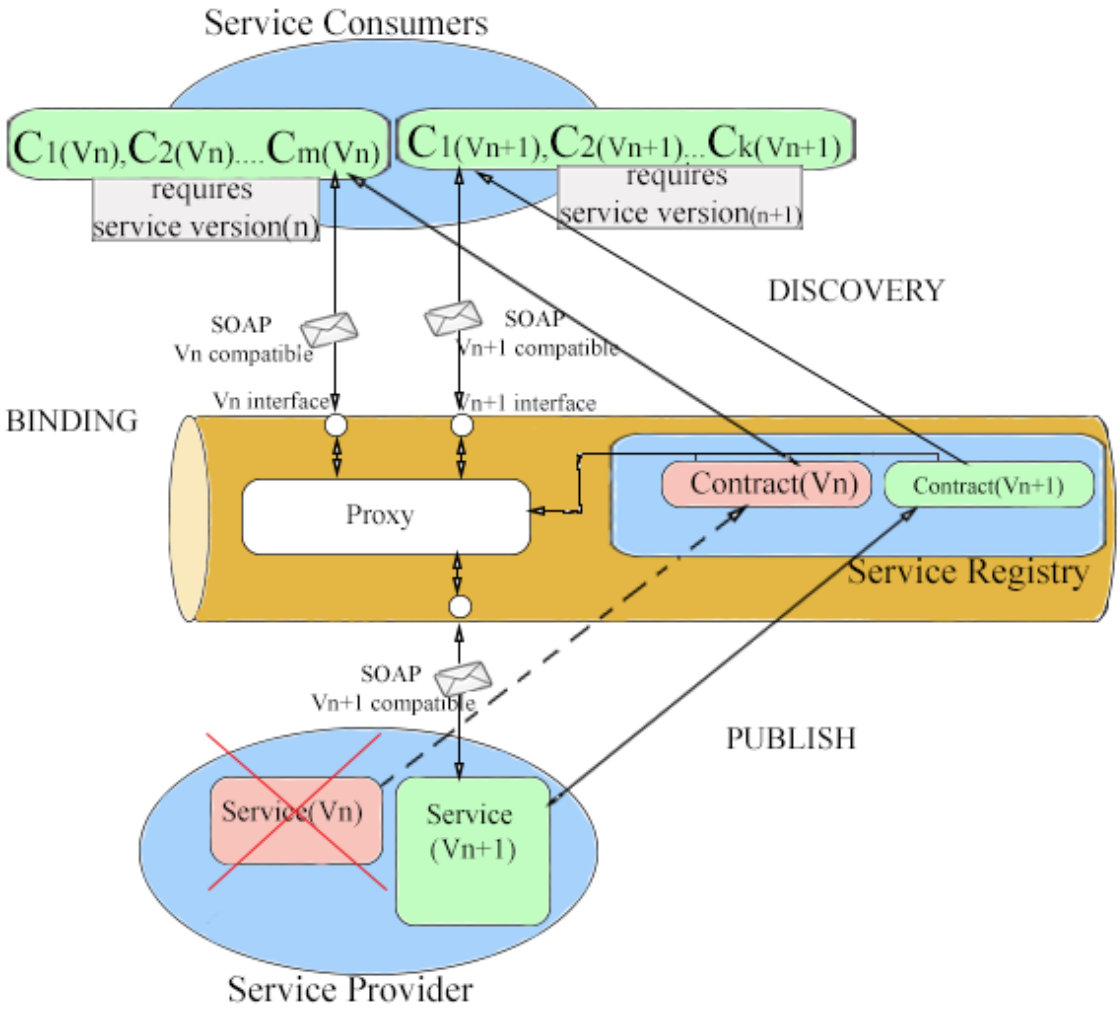

Figure 4.4: The contracts-based proxy for web service management model (Chiponga et al., 2014b).

A. Service Provider

In Figure 4.4, the service provider refers to a system, an organisation or an individual that provides a service by exposing it to other organisations or individuals such that they can consume it over the Internet. The service provider owns the web service and implements the necessary technologies on which the web services they offer are deployed. They provide the web service implementations, publish the service descriptions (contracts), and maintain the web services .i.e., provide the technical support for business processes. The services that the service provider offers are reusable

components representing a complete business task such as a weather lookup, a StockQuote lookup or a credit-card validation. The full descriptions of the capabilities of the service are published in a service registry.

B. Registry

The registry in Figure 4.4 allows businesses to find each other and facilitates communication by hosting a catalogue of published and available web services. The registry uses the Universal Description, Discovery and Integration protocol as the standard method for providers to publish and consumers to discover network-based software components of SOA. The main purpose of the registry is storing and representing the data and metadata of web services. There are private and public registries, and in this work a public registry was used, in which the eventual consumers of the web services published by the service provider are not known. A public registry may appear to the end-user as just a service in the cloud and access to the registry data is open to the public. In Figure 4.4, the registry is the catalogue in which the service provider publishes the technical contracts of the available services.

C. Consumers

A consumer is a member of the set of users of the web service. The consumer can be a web application, software application, mobile application or another web service that requires a service. The consumer is the entity that initiates the discovery of a contract in the registry. They bind to the service upon accepting the contract. The consumer invokes the service by sending a service request formatted according to the contract.

D. The contracts-based model

The consumer described in item C can be a mobile application or a web-based application running through a website. With reference to the running scenario as presented in Chapter 3, Section 3.3, the web application is viewed through a web-browser by the customer to the hotel booking service. Upon submission of a form from the customer, the PHP code creates a SOAP request message and sends it to the Internet address of the web service in the proxy. The proxy receives the SOAP request and opens to read it in order to ascertain which consumer is requesting the service and what service version the consumer is based on. The proxy collects the contract matching that of the identified consumer and the contract of the running StockQuote web service to verify the differences in compatibility between the two contracts. Using the intelligence built into itself the proxy transforms the incoming request from the consumer to match the expected input request of the running StockQuote web service. The proxy then routes the transformed request to the actual Internet address of the StockQuote web service for processing. The web service responds with the appropriate SOAP Response to service the consumer and the response is sent through the proxy in the model, which performs the necessary transformations and routing again on the response. The consumer, the PHP code, receives the SOAP response and proceeds to display the stock values as quoted to the customer. The customer can then proceed to declining or accepting the quote, and continue to book the hotel they wish to stay in.

4.4.3. The Proxy

The main proxy operations as shown in Figure 4.5 are SOAP message identification and SOAP message transformation. Message identification in this case refers to picking out attributes in the incoming message and matching them to an expected set of attributes found in the published contracts to identify which version of the web service is being requested. Once that operation is

complete, then the transformation to match the incoming message to the expected message for that latest running web service is done using a transformer.

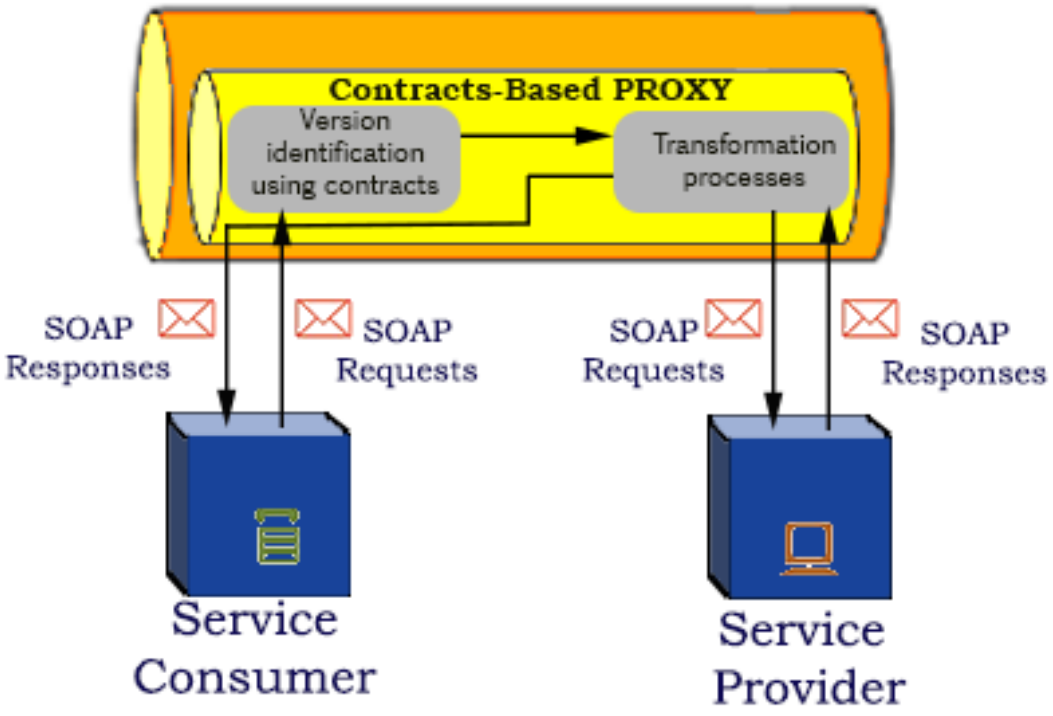

Figure 4.5: The main proxy functions

To achieve these operations the proxy uses the algorithm shown in Figure 4.6. This algorithm compares incoming SOAP request formats with the contract of the running service versions' requirements. This comparison is the basis for the identification of incoming SOAP messages to determine which web service version is being requested and which web service consumer version to service. If the SOAP request format is not a match then the proxy transforms the request to the expected format matching the requirements. The transformed request is then passed on to the actual service for processing. The proxy is targeted for when the service provider does not wish to keep more than one service running in parallel as this will tie up resources that could otherwise be in use for other purposes. Running more than one service in parallel also brings in unnecessary

technical overheads in terms of monitoring, maintenance and troubleshooting. In the algorithm Service(V_{n+1}) and Contract (V_{n+1}) are taken to represent the current (most recent) web service version and web service contract respectively, while Service(V_n) and Contract (V_n) are taken to be the older versions of the web service and contract respectively.

1) The proxy is implemented in this work as a web service which is made available in a service bus to allow for integration. It exposes one endpoint through which all incoming requests will arrive and be accepted for processing. When the proxy is started it reads the contracts that have been manually copied from the registry and saved in a local folder on the host server. These contracts contain all the information and version identification that will be needed by the proxy to identify web service requests. All the contracts reference the same endpoint which is the proxy's endpoint
2) When a consumer is created after accepting a contract, the consumer is regarded as compatible with the web service version that it requests. The consumer invokes the web service through sending a SOAP request message to the message endpoint. The message is received by the proxy for processing
    a. Inside the proxy, upon receiving an incoming message, the proxy has to check and identify the version being requested in order to route it to the correct transformation channel. By checking the parameters that are being requested in the SOAP request the proxy compares that with the expected number of parameters in each contract to find a match. The compatible contract identifies the service version that is being requested and that tells the proxy which path to route the SOAP request
    b. Once the SOAP request has been directed to the correct transformer, the transformer reads the request and creates the new SOAP request using the values

that were in the original incoming request. The transformer creates a message that is compatible with the contract of the implemented web service version

c. The transformer forward the newly created SOAP request to the endpoint of the implemented web service version which is the latest version and waits to receive the SOAP response that will be returned by the web service. If there is any error in invoking the web service, such as no service running, caused by the service being temporarily down for any reason, the transformer returns a SOAP Fault message to the calling consumer

d. After processing, the invoked web service returns as a SOAP response which is received by the proxy. The proxy is already aware of the version of the consumer that invoked the web service, so it converts the SOAP response to the format matching the contract that the consumer accepted. This process uses transformation through a transformer to do the necessary transformations

e. Once the conversion is done to match the contract version that was used by the consumer the SOAP response is sent back to the invoking consumer

3) The default web service version that is configured to respond to incoming requests is the most recent version. This is because the most recent web service is evolved and up-to-date with the expectations of all the consumers and administration of the web service.

a. Hence any incoming SOAP request unmatched to a previous web service version is assumed to be a request for the most recently implemented web service. No transformation is applied. If that request does not match the requirements of all the contracts then a SOAP Fault message is returned to the calling consumer

b. A SOAP response from the web service is related to the calling service, and no transformation is applied as the response already matches the most recent contract

1. **If** not-exists then cache the contracts of the services
2. **If** a request from consumer requires Service(V_n) send to proxy:
    a. **Get** SOAP request and compare format with Contract(V_{n+1}) requirements
    b. **Identify** the service version being requested
    c. **Apply** Transformation to match Service(V_{n+1}) format
    d. **Forward** request to Service(V_{n+1}) and Listen for response
    e. **Get** the SOAP response and apply Transformation to match Contract(V_n) response
    f. **Send** response to consumer
3. **Else** send request to Service(Vn+1)
    a. **Listen** for response from Service(V_{n+1})
    b. **Send** response to requesting consumer

Figure 4.6: The service proxy algorithm listing (Chiponga et al., 2014b)

This algorithm in Figure 4.6 allows for the identification and transformations of incoming SOAP requests from the consumers from any number of versions of the web service. For easier management and demonstration purposes, these consecutive versions were limited to three in this work. Change processes and organisational policies can be put in place to recommend how many virtual versions should be supported by the proxy. The introduction of a new version means minor reconfigurations in the proxy. There are two possibilities that were identified. The first is the introduction of a new web service version while maintaining the same number of supported web service versions. The first possibility assumes that the organisational policies says for example that

at most three web service versions are to be supported at the same time. This would mean the organisation has put out a notice to all its customers that all support for the oldest version of that web service will cease on a particular date and time. To replace a web service version that was currently being supported by the proxy, there is no reconfiguration of the proxy that is required. The new contract will need to be uploaded to the container where the other contracts reside, and the matching to the new web service version in the corresponding transformers will need to be updated. The proxy will then need to be restarted for the updates to be activated.

The second possibility is the introduction of a new web service version, increasing the number of supported versions while maintaining the already existing support for older versions. This would be the case if say the organisational policy on the maximum number of service versions to be maintained is more than the currently supported versions. Extending from the already three supported web service versions to four implies the addition of a transformation path to the choice router in the proxy. The new contract will need to be uploaded to where the rest of the contracts are located, and a new transformer will be added in the new path that is needed to support the new web service version. As in the first possibility, all the transformation files associated with the transformers for each transformation path will need to be replaced with an updated one that maps to the new web service version. In this instance however, the proxy will need to be recompiled and restarted for the updates to be effected.

4.5. Summary

This chapter discussed the design of a contracts-based proxy model for managing web service evolution. The model was presented in this chapter as the artifact aimed at resolving some of the challenges in the maintenance of evolving web services. Best practices in software configuration management, the principles of evolution and best practices in contract design were incorporated

into the design of the model. The overall design of the model theoretically supports any number of web service versions without disrupting service to the consumers of the older versions.

Chapter 5 validates the model that was designed in this chapter. Furthermore, Chapter 5 describes an instantiation of the model as a proof-of-concept prototype.

CHAPTER FIVE

5. MODEL VALIDATION AND PROOF-OF-CONCEPT PROTOTYPE

In Chapter 4, the contracts-based model was designed. System development resulted in an artifact which was the proof-of-concept for the fundamental research that was done. The artifact is the model construction. Design science (Hevner *et al.*, 2004) seeks to validate the result in order to ascertain the artifacts' applicability in a real world environment. Thus in this chapter, the goal is to validate the proxy-based web services evolution model that was presented in Chapter 4. The validation in this work was done in three stages:

1) Stage 1 was the validation of the Model's efficacy through professional consultations
2) Stage 2 was demonstrating the practical utility and applicability of the model through the use of the running scenario described and used in the previous chapters
3) Stage 3 demonstrated the technical feasibility by means of proof-of-concept prototyping and experimentation while replicating the theoretical results

Since the 2nd type of validation has been demonstrated through the running scenario in Chapters 3 and 4, this chapter briefly discusses the 1st type of validation then focuses on the realisation aspects of the prototype, which is the 3rd type of validation.

5.1. The Proof-of-Concept Prototype

The 1st type of validation was achieved through consultation. With the design criteria in mind, and both Lehman's Law of Software Evolution and the best practices in Software Configuration Management a model targeted at managing the evolution process of a web service using contracts was designed. For the purposes of continuous improvement and to ensure applicability to evolution management of web services the proposed model was subjected to the critique of experts in the field of web service management. The objectives for exposing the proposed model to professionals in the field included:

- Determining a range of possible program alternatives and solutions
- Exploring the underlying principles, theories and assumptions leading to different conclusions or judgements
- Correlating informed judgements on a topic that may be spanning a wide range of disciplines and
- Educating the respondent group on the aspects of the topic in question

To achieve the aforementioned objectives, the proposed model was sent to peer reviewed conferences to obtain views and comments with regards to the efficacy, utility and applicability of the model. The model was exposed to a total of 9 reviewers, who contributed to the refining of the model and extended the thinking around the same. A concern on how the model caters for a situation where a web service completely evolves leaving no resemblance of the previously existing versions was raised. This concern aided the research presented in this work, to include and understand how the Laws of evolution help clarify that. In evolution, once all familiarity with a previous version is lost, it can no longer be classified as evolution but replacement [of a web service]. The contracts-based proxy has an impact on the overall performance of the web service,

relative to the client applications. This was further explored with the aid of some of the experts to focus on the specific performance metrics that would need further investigation. Hence, Chapter 6 details the performance metrics and evaluations on the proposed model instantiation to investigate impact of the contracts-based proxy model. With each new service feature/upgrade, the proxy needs to be maintained to reflect the same. Some discussions around maintenance of the proxy as updates are rolled in and new service versions are implemented, were brought up. This further strengthened the advantage that comes with having the contracts-based proxy compared to hosting multiple versions or forcing customers to upgrade their consumer applications because development and redeployment costs can soar out of control, depending on how often upgrades are introduced. Thus updates will mean a little extra work on the part of the service provider's development team but a more reliable service with respect to consumers, hence maintaining evolution of the web service in a more controlled manner. All in all, consultations with professionals in the field helped shape and refine the model to what was presented in Chapter 4 – the contracts-based proxy for web service evolution management.

5.1.1. The Experimental Setup

For the 3rd type of validation free and widely supported open source software tools like apache-tomcat were chosen. The underlying technologies in the implementation of this open source software are common standardised software tools used in the industry and used by most developers from all over the globe. Figure 5.1 is illustrating the setup that was implemented in contribution to the validation of this model.

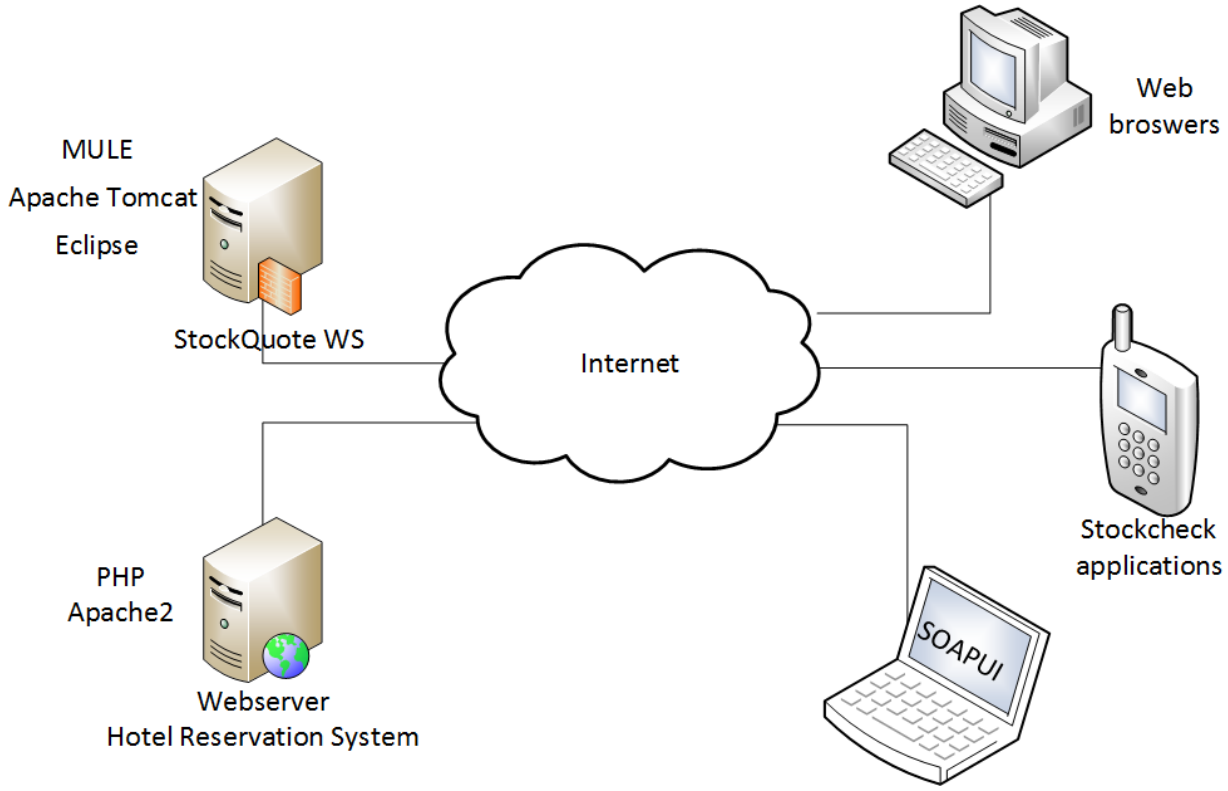

Figure 5.1: The experimental setup and technologies employed.

Web services are platform independent. To host the StockQuote web service a server running the Windows operating system was configured, whose specifications are tabulated in Table 5.1. Although the minimum requirements for setting up an environment to program and develop web services are as low as 512MB of memory and Windows 2000/XP professional operating system with a Pentium 3 processor, a server machine with high specifications was selected. The reason for such a selection was in an effort to limit or eliminate if possible, the hardware impedances in the performance of the proxy due to machine processing capabilities. On the server-side, Eclipse as the IDE and apache Tomcat 7 were configured to host the implementation of the StockQuote web service. MuleESB was installed and configured to host the implementation of the contracts-based proxy that would manage the incoming requests for the StockQuote services. This is described in more detail in the implementation section of this chapter, Section 5.2. The StockQuote

in this work's experimentation had three variants representing the three versions of the web service that has evolved from version 1.0, through 1.1 to 1.2.

Table 5.1: The server machine specifications.

Processor:	Intel Core i7 – 4500U, 1.8GHz 2.40GHz
Hard Drive space:	1Tb
Memory:	8GB (7.89GB usable)
OS:	Windows 8 Single Language
System Type:	64-bit OS, x64-based processor
Other :	USB, DVD support

In the experiment, the Hotel Booking Service was set up on a Linux box, and written in PHP as one of the consumers of the StockQuote service. Consumers / Client devices access the services over the Internet, and in the experimentation two clients were used to invoke the StockQuote in the local area network. SoapUI was used to mimic a large number of clients invoking the web service randomly from a laptop on the network. The web services used in the evolution scenario to test the usability and applicability of the proposed model were implemented in a closed laboratory environment across a local area network to simulate the Internet.

5.1.2. Underlying Technologies

Web services are known to provide a language-independent and platform-independent infrastructure for the integration of heterogeneous components. After having chosen the architecture in which lies the context of this work, the task was to choose the platform capable of building and supporting SOA. Given that the goal of this work was not to reinvent the wheel by

building a new set of stand-alone web services that would be used to investigate the proposed model, a set of services that were made available for teaching and research in SOA were employed. Though the original services were implemented using PHP, implementing some parts of the system proved challenging as the updated versions of the software environments were incompatible with the original configurations and coding. But due to the platform independence and loose coupling feature of web services, the StockQuote functionality was implemented and integrated using java. For the purpose of illustrations, Eclipse, apache tomcat, apache axis2, and the ESB from Mulesoft (Mule-ESB) were used to build the service environment and SoapUI to simulate the clients.

5.1.3. Eclipse

Eclipse Juno build 20130225-0426 (on jdk1.8.0.5) is the development environment which was set up. Eclipse provides a universal toolset for developers mostly in but not limited to java. It is an open-source integrated development environment (IDE) (Guindon, n.d.), whose development language support is independent and not limited in any way. It is also the most widely used open-source IDE.

5.1.4. Apache tomcat

The highest version of the suitable server runtime compatible with Eclipse Juno at the time of implementation was Apache Tomcat v 7.0. Apache Tomcat is also an open-source software like Eclipse. It is developed in a participatory environment in an effort to foster collaboration between the best of developers from around the globe and is developed under the Java Community Process. Apache Tomcat is behind the numerous mission-critical, large-scale application across the globe and a diverse range of organisations, industries and personal developers.

5.1.5. *Apache Axis2*

Axis2 web services core v1.1 was used as it enables web services generation through the web services engine in Eclipse using the dynamic web module 2.5 facet of the target apache tomcat runtime environment. Apache Axis2 is the most popular and widely used core engine for web services because it is more XML-oriented, and more efficient and modular compared to Axis. It was carefully designed for easy plugin modules to extend its functionality for security and reliability features.

5.1.6. *Mule ESB*

Mule ESB is a lightweight java-based ESB and integration platform. Developers can connect applications easily allowing for the exchange of data among applications. Mule ESB allows different applications irrespective of the technologies used (JMS, HTTP, REST, Web services, and JDBC) to communicate across an enterprise or even the Internet.

The contracts-based proxy was implemented in the ESB. MuleESB provided a visual / graphical programming environment with drag-and-drop components which could be easily and quickly configured. The components were customised to achieve the logic of the proxy as described in Chapter 4, Section 4.4.3 of this work. Custom XSLT files were written and incorporated into the proxy to perform the required transformations in the ESB.

This set of technologies formed the basis for the prototype implementation of the contract-based proxy for web service evolution management. Section 5.2 describes the implementation of the prototype that was developed.

5.2. *The Prototype Implementation*

To realise the prototype, this work started off by designing the main and most important component of the functionality of the proxy. The main component in this model is the contract. The contract describes the web service's offerings and the expectations of any consumer that chooses to use the service. DbC is regarded as the best approach to designing web services and it is a design pattern that focuses on creating the WSDL file first. In SOAP web service development, the WSDL is the most important and essential component for presenting the web service contract. The contract structure, written using XML, is defined and can be extended by the use of other technologies such as XSD or XML Schema and Web Service Policy Language (WS-Policy) as shown in Figure 5.2.

XSD provides a language defining the validation constraints of messages in XML format. XSD is also written in XML and thus seamlessly integrates with the contract either by embedding it in the contract or by having the XSD being kept as a separate file. Separating the WSDL, XSD and WS Policy files further enhances decoupling in SOAP web services.

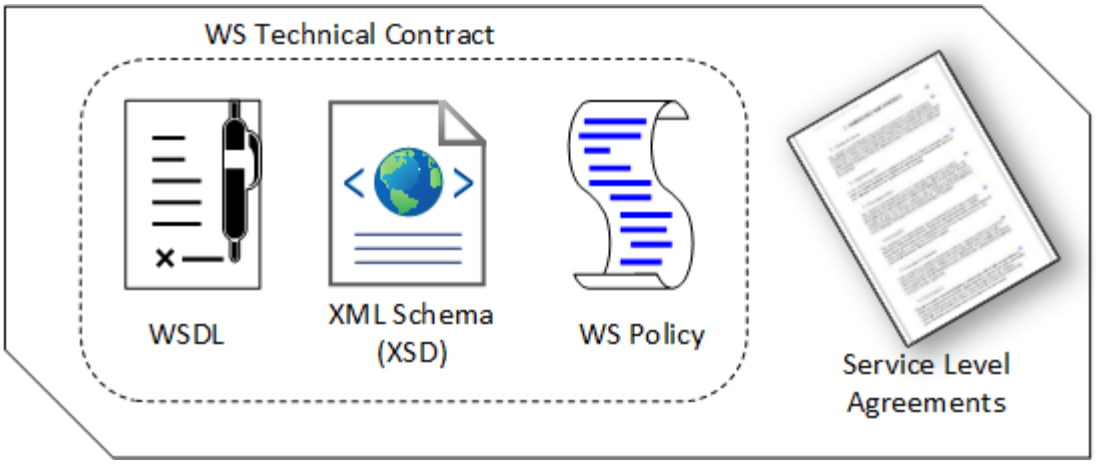

Figure 5.2: Graphical view of the Web service contract (Erl et al., 2008).

The decoupling enables easy readability of the service contract for the developer as they are designing the web service. The WS-policy is also expressed in XML and specifies behaviour

related constraints of the web service supplementing (adding to) the definitions in the descriptions of the contract. As illustrated in Figure 5.2, Service Level Agreements can be part of a web service contract but are outside the technical contract that is used by machines, and hence are not included in the implementation of this work.

XML is the markup language that is used to write the web service's contract. Figure 5.3 shows the contract for our StockQuote web service version 1.0 expressed using XML.

Figure 5.3 is organised into the basic structure of a contract showing what the purpose of the StockQuote web service is, how the StockQuote web service can be accessed and where the StockQuote web service is located. This structure is what the consumer needs to understand in order to successfully invoke the web service.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tns="ht
  <wsdl:types>
    <xsd:schema targetNamespace="http://lapnet8.uzulu.local/sq_v1.0/">
      <xsd:element name="StockQuote" type="tns:stockQuoteRequestType">

      </xsd:element>
      <xsd:element name="StockQuoteResponse"
        type="tns:stockQuoteResponseType">
      </xsd:element>
      <xsd:complexType name="stockQuoteRequestType">
        <xsd:sequence>
            <xsd:element name="stockSymbol" type="xsd:string"></xsd:element>
        </xsd:sequence>
      </xsd:complexType>
      <xsd:complexType name="stockQuoteResponseType">
        <xsd:sequence>
            <xsd:element name="price" type="xsd:float"></xsd:element>
            <xsd:element name="AdditionalInfo" type="tns:AdditionalInfoType1"
            minOccurs="0"></xsd:element>
        </xsd:sequence>
      </xsd:complexType>
            <xsd:complexType name="AdditionalInfoType1">
                <xsd:sequence>
                    <xsd:any minOccurs="0" maxOccurs="unbounded"></xsd:any>
                </xsd:sequence></xsd:complexType>
      </xsd:schema>
  </wsdl:types>
  <wsdl:message name="StockQuoteRequest">
    <wsdl:part element="tns:StockQuote" name="parameters"/>
  </wsdl:message>
  <wsdl:message name="StockQuoteResponse">
    <wsdl:part element="tns:StockQuoteResponse" name="parameters"/>
  </wsdl:message>
  <wsdl:portType name="sq_v1.0">
    <wsdl:operation name="StockQuote">
      <wsdl:documentation>version:1.0</wsdl:documentation>
      <wsdl:input message="tns:StockQuoteRequest"/>
      <wsdl:output message="tns:StockQuoteResponse"/>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="sq_v1.0SOAP" type="tns:sq_v1.0">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="StockQuote">
      <soap:operation soapAction="http://lapnet8.uzulu.local:8080/stockquote/StockQuote"/>
      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="sq_v1.0">
    <wsdl:port binding="tns:sq_v1.0SOAP" name="sq_v1.0SOAP">
      <soap:address location="http://lapnet8.uzulu.local:8080/stockquote/StockQuote/"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Figure 5.3: The code listing for StockQuote version 1.0

5.2.1.1. Contract Port-type

The port-type is the access point for the web service. One can think of the port-type as an international airport on an island. All the incoming and outgoing people pass through the airport and without it they would have nowhere to land so ultimately no entry or exit point. A port-type contains the set of operations exposed by the service for example, lines 35-41 in Figure 5.3. The contract can have one or more ports-types.

5.2.1.2. Contract operation definition

The operation is the function that is exposed to the consumers for what the web service can do. The operation definition is where the message definition is found. The message definition defines the data that will be transmitted between consumer and provider. Lines 38 and 39 in Figure 5.3 show that the expected message will be a SOAP message. Message definitions could be one of three types: input, output or fault message. The StockQuote implementation exchanges the input of a stock symbol and returns a StockQuote thus implementing the first two types of message definitions.

5.2.1.3. Endpoint and address bindings

The address binding shows where the web service is located. The address binding is a pointer to the actual network address where the web service can be accessed; Lines 42 and 43 in Figure 5.3. The endpoint is the container for the address binding. The same endpoint can be used for different operations, different message bindings and different port-types thus enabling us to use the same endpoint for different StockQuote web service versions implemented for the purposes of testing this prototype instantiation.

5.2.1.4. Data exchanged

According to the contract for the StockQuote version 1.0 in Figure 5.3 the consumer is expected to supply a correctly formatted SOAP request with the requesting element name StockQuote. The request should contain the stock symbol which is a string; Line 13 in Figure 5.3. The consumer expects a response from the StockQuote service which is the value of the stock at the time of the request. The response named StockQuoteResponseType, will be the float value of the stock on the day; Line 18 in Figure 5.3. The implementation of this contract was designed with evolution in mind as is evident in Figure 5.3, line 25. The "xsd:*any*" element is optional as indicated by the *minOccurs* value of 0, and can be used for additional information when needed. This means that this element can be used in a future version where the consumers need more data from the web service. The number of occurrences is unbounded and thus the service in a newer version can use this element without having to redefine the contract and avoiding disrupting the consumers already running with the version 1.0 contract.

5.2.1.5. The StockQuote web service

In an effort to demonstrate the efficacy of the proposed model, the first version of the StockQuote web service implementation was designed as illustrated in Figure 5.4. In summary, Figure 5.4 shows the business process implementation of a live StockQuote version 1.0 which was built to demonstrate the evolution of the web service. The change and evolution scenarios are fully described in Chapter 3, Section 3.5 where the StockQuote web service evolves from version 1.0 through to version 1.2. The StockQuote web service version 1.0 in Figure 5.4 receives a stockSymbol from the requestor and, in turn, calls on another web service and receives the stock values. It then prints the resulting quote and returns a response to the consumer.

```
1   package local.uzulu.lapnet8.sq_v1_0;
2   import net.webservicex.*;
3   import com.eoddata.ws.*;
4   import com.cdyne.ws.*;
5   /**
8   public class Sq_v10Skeleton implements Sq_v10SkeletonInterface{
9       /**
14  public local.uzulu.lapnet8.sq_v1_0.StockQuoteResponseDocument StockQuote
15                  (
16                  local.uzulu.lapnet8.sq_v1_0.StockQuoteDocument StockQuote0
17                  )
18          {
19              //the business logic starts here
20              String stockSymb = StockQuote0.getStockQuote().getStockSymbol();
21             //just a system message to log when service is invoked
22                  System.out.println("Recieved "+ stockSymb + " as the input symbol from consumer");
23                  StockQoteSOAPQ1Stub quote1 = (StockQoteSOAPQ1Stub)new StockQurey().getNewQuote(
24                  new URL("http://ws.cdyne.com/delayedstockquote/delayedstockquote.asmx"));
25                  StockQoteSOAPQ1Stub quote2 = (StockQoteSOAPQ1Stub)new StockQurey().getNewQuote(
26                  new URL("http://www.webservicex.net/stockquote.asmx"));
27                  StockQoteSOAPQ1Stub quote2 = (StockQoteSOAPQ1Stub)new StockQurey().getNewQuote(
28                  new URL("http://ws.eoddata.com/data.asmx"));
29                  System.setProperty("http.proxyHost", "studproxy.uzulu.ac.za");
30                  System.setProperty("http.proxyPort", "3128");
31
32                 QuoteReturn bankQuote1 = quote1.GetQuote(stockSymb,0);
33                  QuoteReturn bankQuote2 = quote2.GetQuote(stockSymb);
34                  QuoteReturn bankQuote3 = quote3.QuoteGet(noLogin,Currency,stockSymb);
35                  //create the SOAP response and attach the best quote
36
37                  StockQuoteResponseDocument outDoc = StockQuoteResponseDocument.Factory.newInstance();
38                  StockQuoteResponseType outType = StockQuoteResponseType.Factory.newInstance();
39
40                  float quotePrice = (float) quote.getMin(bankQuote1, bankQuote2, bankQuote3);
41                  outType.setPrice(quotePrice);
42                  outDoc.setStockQuoteResponse(outType);
43                  //system message as confirmation
44                  System.out.println("We are restuning the value of :R" + quotePrice +" As the total quoted");
```

Figure 5.4: The code listing for the StockQuote web service version 1.0

Figure 5.5 shows the available StockQuote version 1.2 that was deployed in the setup. In deploying the most up-to-date web service, StockQuote version 1.2, a dynamic web service was created in eclipse and the web module was set to version 2.5, which is required by Axis2. The target runtime environment was Apache Tomcat version 7.0. The configuration was set to custom settings in order to select and enable Axis2 web services core 1.1, which prepares the project for Web services generation through the Axis2 Web services engine.

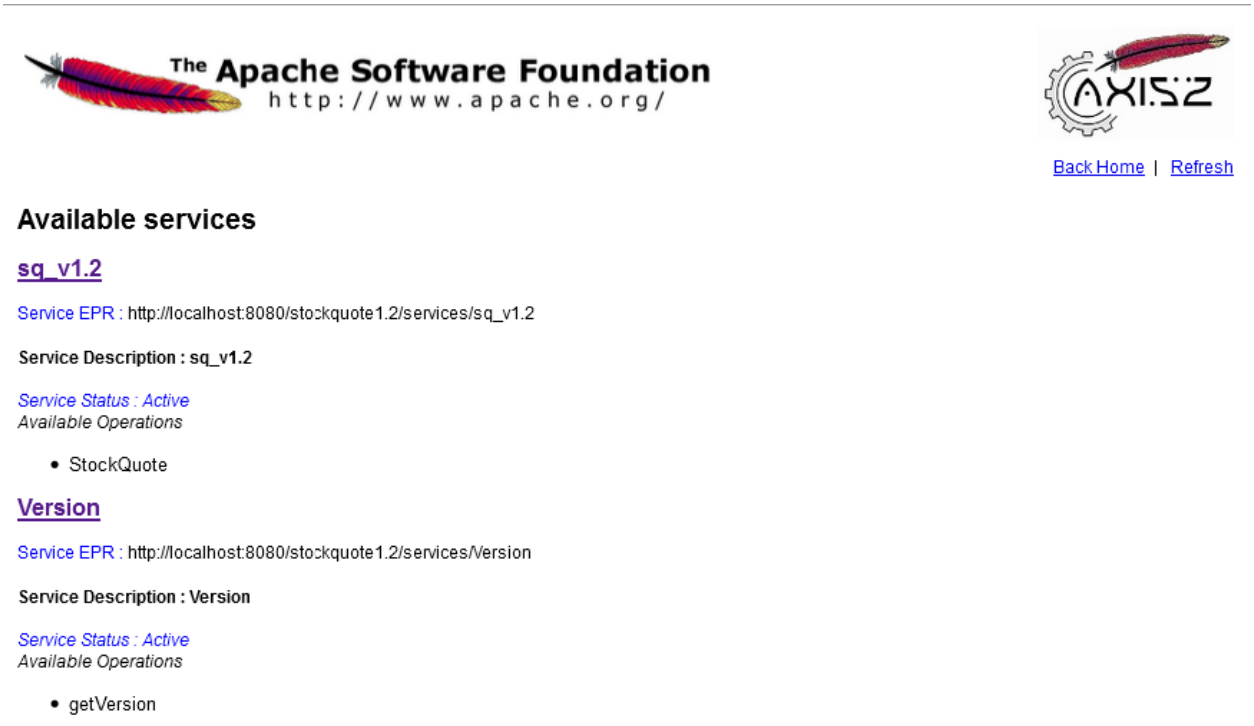

Figure 5.5: StockQuote version 1.2 deployed in tomcat and Axis2

Figure 5.6 shows the process of creating the StockQuote web service. Having designed the contract following the DbC approach the wsdl2java tool was used to create the default skeleton code that is required to implement the web service. The business logic of the updated StockQuote web service version 1.2 was implemented like that in Figure 5.4 by modifying the skeleton code. The generated Service Endpoint Interface (SEI) plus the business logic that was implemented are collectively known as the source code of the StockQuote web service as depicted in Figure 5.6.

Once the business logic was written and “dry-runs” were performed to test the logic, the code was compiled using the “ant” compiler. The compiled code was packaged and deployed in the Tomcat container resulting in the web service that is depicted in Figure 5.5.

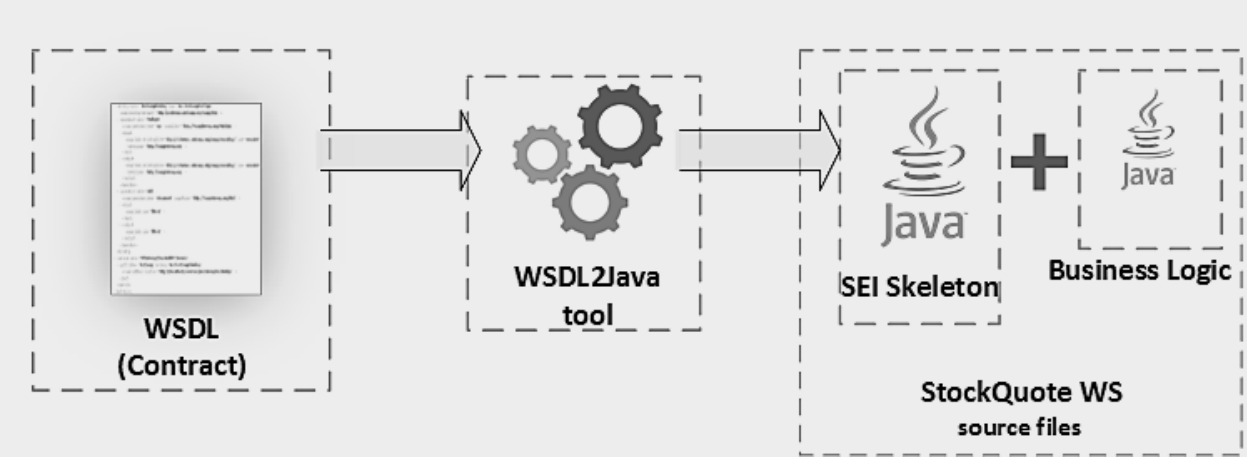

Figure 5.6: Generating code with the WSDL2Java tool

At version 1.0, the service returns a price of the stock symbol upon being invoked. At version 1.1, the service can also return additional information on the previous day’s highest and lowest value of the stock symbol. This is after a consumer has provided the optional parameter to request the additional information for a real-time quote. At version 1.2, the web service is now more enhanced to accept a third parameter requesting earnings. This enhancement is on top of the already existing capabilities of the two previous versions. Version 1.2 was carefully designed following the DbC approach, while keeping in mind the Laws of evolution and SCM best practices. This careful design enable version 1.2 to support requests from older service consumers. The contract is backward compatible in that it has the capability to support the old consumers but the problem that comes up is that the implementations of the old consumers would need to be updated in order to use the latest version of this service; hence, the implementation of the proxy to address challenges

faced in upgrading. This will be suitable for application in web based financial systems where you do not want to break any service because *"the stakes are too high in the financial context"*, according to the interviewed developers (Espinha *et al.*, 2015).

5.2.2. The proxy implementation

To implement the proxy, the host environment chosen was MuleESB. MuleESB exhibits properties that include message routing, transport management, transaction management and security. Mule is capable of supporting different vendor implementations, can connect multiple applications at the same time, and is highly scalable. The reasons behind choosing MuleESB over other competitors include the fact that Mule allows for the use of existing components without changes improving component reuse, components do not require Mule-specific code to work with Mule, and messaging logic is kept completely separate from the business logic. MuleESB does not limit design to any specific architecture and is lightweight, reducing time-to-market for projects and increasing productivity.

Implementing the contract-based proxy in MuleESB gives us access to all the built-in advantages already implemented, allowing focus to be placed on the core functionalities needed in managing the evolution process of a SOA. To build the proxy, the already existing components in Mule were assembled and reconfigured to suit the purposes for which they were needed in the script as, according to the design criteria, much of intelligence needed to achieve the routing capability and scalability is implemented in the MuleESB software - which has recently gained popularity and become part of the architecture in many SOA implementations.

The service proxy algorithm discussed in Chapter 4, Section 4.4.3, was used as the template for the logic that was followed in implementing the contract-based proxy service. In MuleESB, the

proxy was deployed as a web service performing service mediation, shielding the running web service from receiving incorrectly formatted requests and shielding the consumers from receiving a message response incorrectly formatted for a different contract than the one they are expecting. To achieve the functionality following the algorithm, the design advertises a single endpoint, *<soap12:address location="http://localhost:8081/StockQuote"/>*, which is advertised in all contract versions published in the registry in the model, i.e., all contracts have the same endpoint so that all consumers are directed to the same proxy that is running on the published port. Once the proxy is up and running it receives messages from consumers, extracts the body of the incoming SOAP request and sets the body as the payload to be used for consumer version identification. The payload is forwarded to the secondary function of the proxy.

A secondary function of the proxy is to detect the service version on a SOAP request and to determine which transformations path the request should be sent through as shown in Figure 5.8. After a SOAP request is received by the proxy, the proxy checks the service version of the incoming request by comparing the supplied parameters in the request to the expected parameters in the published service contracts to determine which service version is being requested. Once the version being requested is identified, the SOAP request is forwarded to the primary function of the proxy. The XML in Figure 5.7 represents the choice flow control and criterion used to determine the transformation path a request will be sent through.

```
<choice doc:name="Choice" tracking:enable-default-events="true">
    <when expression="#[xpath('/StockQuote/addInfo').text==&quot;true&quot;]">
        <flow-ref name="mainBusinessLogic" doc:name="LatestVer"/>
    </when>
    <when expression="#[xpath('/StockQuote/realTime').text==&quot;true&quot;]">
        <flow-ref name="transformForV1.1" doc:name="Ver1.1"/>
    </when>
    <otherwise>
        <flow-ref name="transformForV1.0" doc:name="Ver1.0"/>
    </otherwise>
</choice>
```

Figure 5.7: The code listing for the Choice for transformation path

The primary function of the proxy is the transformation of the SOAP request to meet the expectations of the implemented service version. In the model proposed, the most recent version of a service is the one maintained as it can meet all the needs of both the old and new consumers.

Figure 5.8 shows the examples of the transformation paths for the transformation of a SOAP request to match the current running version of the web service. The transformations in the case of the proposed model are done using XML transformations.

- The choice flow control: this routes SOAP requests on the basis of the properties of the payload. The payload is the body of the SOAP message
- The expression: this is the evaluation criteria to check the content of the message. Using XPATH, the tree structure of the SOAP message is traversed to the existence or lack thereof, of an element: this evaluates to true or false allowing for a decision to be made on the basis of the existence of the property
- The flow-ref: this is the routing path or option taken in the case that the expression has been validated. "*flow-ref: name=”transformForV1.0”*" will imply that the SOAP message will be forwarded to the flow called transformForV1.0 inside the main flow

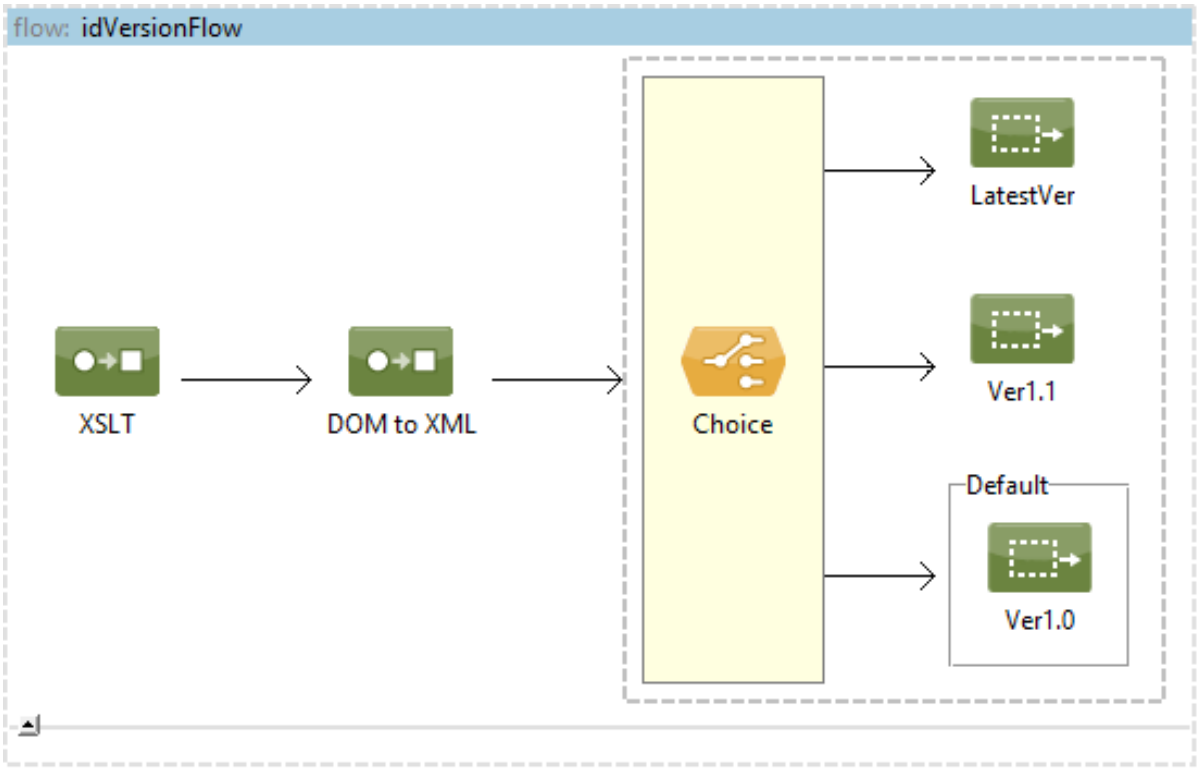

Figure 5.8: SOAP request version identification.

Extensible Stylesheet Language Transformations (XSLT) is a language for transforming XML documents into other XML documents. This is used in this work in conjunction with XPath, an expression language that is fundamental to XML processing. Figure 5.9 and Figure 5.10 show snippets of the XML and the XSLT used to transform an incoming SOAP request for service version 1.1 to meet the contractual expectations of service version 1.2:

- xslt-transformer: in Figure 5.9, the transformer transforms the SOAP message using an XSLT style sheet. The template file used to transform the SOAP request message of service version 1.1 to match the expectation of the contract version 1.2 is shown in Figure 5.10.

```
<mulexml:xslt-transformer maxIdleTransformers="2" maxActiveTransformers="5"
xsl-file="C:\Users\kuku\AnypointStudio\workspace\wsdlistproxyservice\src\
main\resources\incMap4choice.xsl" doc:name="XSLT"/>
<mulexml:dom-to-xml-transformer doc:name="DOM to XML"/>
```

Figure 5.9: The code listing for XSLT transformer and map

Figure 5.10 shows the xml transformation template that is used in transforming incoming SOAP messages. Using the XSLT template in Figure 5.10, the SOAP request is reconstructed to match the expectations of the service contract version 1.2. The result is the transformed SOAP request for version 1.1 that can be used to successfully invoke a version 1.2 service. The template matches the *StockQuoteSymbol* and the *realTime* elements and populates their values using the, *<xsl:value-of select=”.”/>,* values from the incoming SOAP request. The template then adds the *addInfo* element into the SOAP without a value to create the transformed SOAP request. This SOAP request is valid as the contract for version 1.2 was designed with backward compatibility and *addInfo* having been specified as an optional element.

```
1  <?xml version="1.0" ?>
2  <xsl:stylesheet version="1.0" xmlns:soap="http://www.w3.org/2003/05/soap
3      <xsl:output method="xml" indent="yes" omit-xml-declaration="no" enco
4      <xsl:template match="/">
5          <StockQuote>
6              <stockSymbol>
7                  <xsl:apply-templates select="StockQuote/stockSymbol"/>
8              </stockSymbol>
9              <realTime>
10                 <xsl:apply-templates select="StockQuote/realTime"/>
11             </realTime>
12         </StockQuote>
13     </xsl:template>
14     <xsl:template match="stockSymbol">
15         <xsl:value-of select="." />
16     </xsl:template>
17     <xsl:template match="realTime">
18         <xsl:value-of select="." />
19     </xsl:template>
20     <xsl:template match="addInfo">
21         <xsl:value-of select="." />
```

Figure 5.10: The code listing for XSLT transformation template

Figure 5.11 shows the transformation processing of the SOAP request as it passes through the proxy. The XSL tool transforms the request by applying the XSLT template, and the DOM-to-XML tool converts the result of the XSL tool to XML which is mapped by the data-mapper tool

to the expected structure of the request before calling the web service version 1.2 using the web service consumer tool. The SOAP response is subjected to similar transformation to match it to the calling service before the SOAP response is given to the calling client.

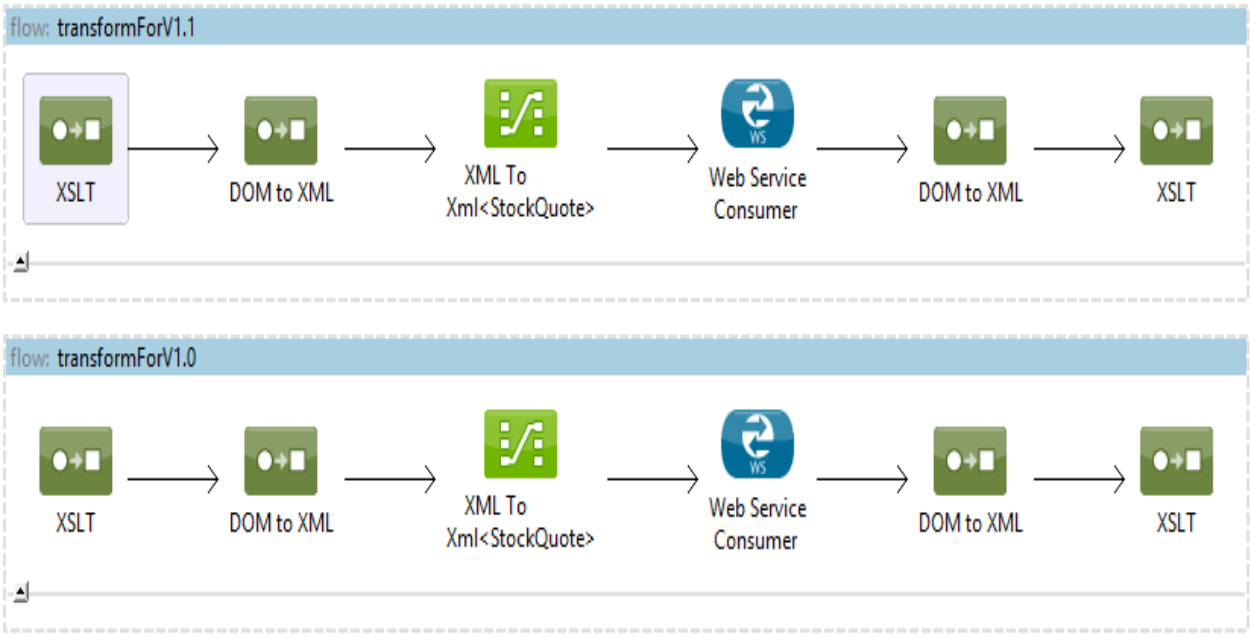

Figure 5.11: SOAP request transformation.

5.2.3. Client simulation and results collection with SoapUI

SoapUI is an open-source web service mocking and testing application for SOAP and REST. It has the function of web service compliance testing, load testing, functional testing, simulation, development, and mocking. This was used to simulate the consumers that invoke the web services. Figure 5.12 shows SoapUI and the SOAP requests for each StockQuote web service version. Each of these requests is sent to the endpoint that belongs to the proxy as specified in the contract and processed as described in the proxy implementation.

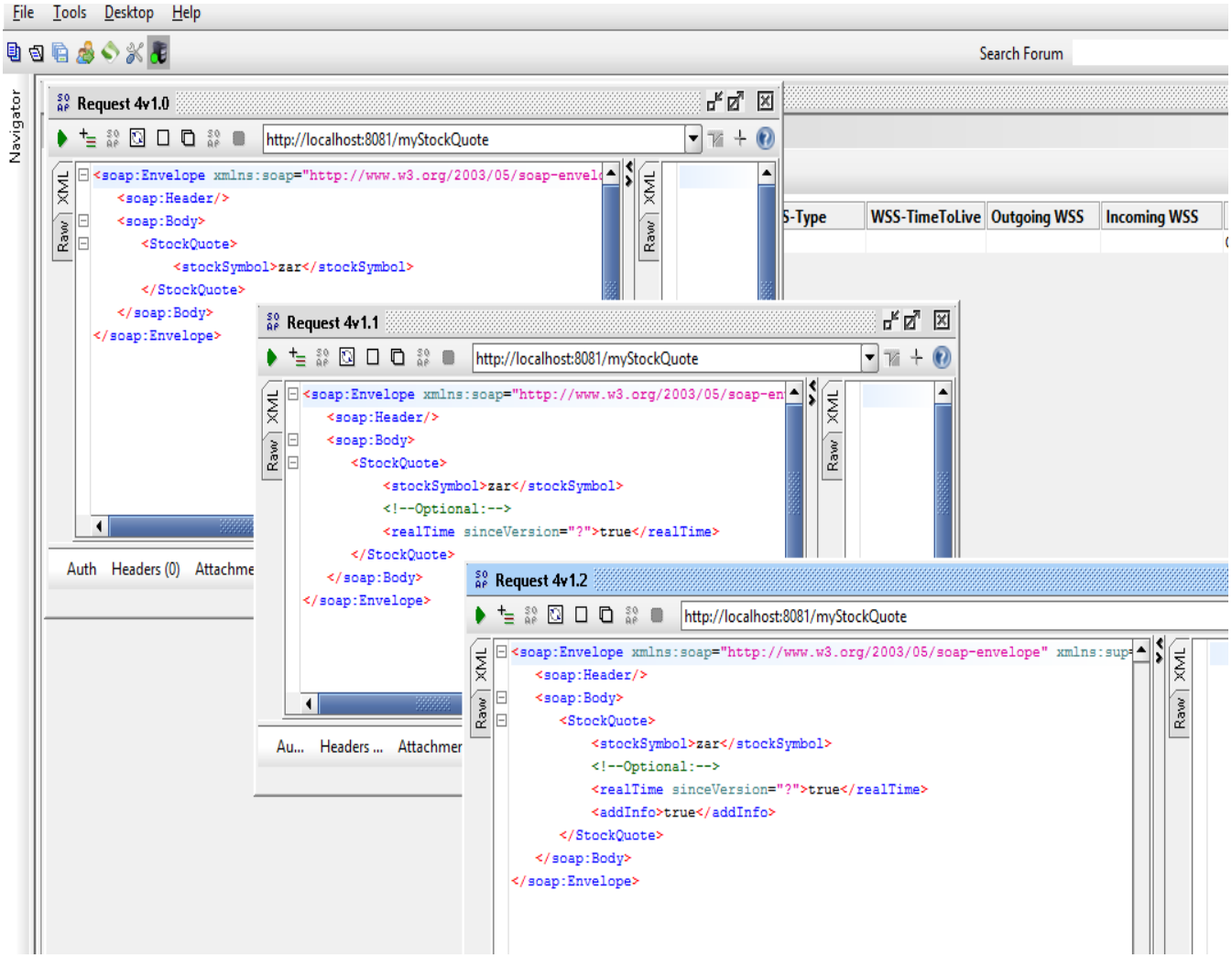

Figure 5.12: SoapUI client simulation for versioned StockQuote web service

Having the most up-to-date StockQuote web service version 1.2 and the contracts-based proxy for web service evolution management in place, consumers that would generate service requests for all the StockQuote web service versions needed to be developed. Consumers based on each contract version that was developed and published could have been created but as the purpose of this work was not in "reinventing the wheel" by creating web services and consumers, a suitable open-source tool was chosen to simulate the consumers and generate the requests for the experimentation. SoapUI is also equipped with the ability to collect the data relative to the simulation of consumers and was used to gather the data presented in Chapter 6.

5.3. Summary

This Chapter discussed the validity of the proposed model by implementing an instantiation of the proof-of-concept. This proof of the technical feasibility of the proposed and developed model complimented the theoretical validity that was discussed at the beginning of this chapter. In the next chapter, results and discussions are presented. Experts in research and SOC were consulted in asserting the need, and consequently the applicability in web service evolution for the contracts-based proxy model for managing web service evolution as presented in Chapter 4. The consultation helped in refining the model design and strengthened the theory behind it. Having established the importance of the model, we set out to further show the technical soundness of the model by means of a successful implementation.

MuleESB was the choice ESB in which to implement the contracts-based proxy for managing the evolution process of the StockQuote web service. MuleESB comes preconfigured with message routing capabilities and was the ideal integration platform, enabling easy message exchange between the StockQuote web service and its consumers. The consumers of the web service virtual versions were emulated using SoapUI, the Swiss-knife software tool for testing SOAP web services.

SoapUI was used to send a SOAP request for a particular service version. The proxy in the ESB successfully detected the StockQuote version that was being requested, performed the correct transformations and successfully invoked the running StockQuote web service in Eclipse. All three StockQuote web service versions were supported successfully through the proxy by only the latest implementation of the web service and in all test instances, the consumer always had the correct and expected SOAP response. It was noted that there was a small increase in the response times due to the processing time required in the proxy, but all three web service versions were

concurrently supported. The concurrent support for all StockQuote versions show that the model can manage the evolved web service without disrupting consumers or requiring that the older consumers be upgraded.

CHAPTER SIX

6. RESULTS AND DISCUSSIONS

According to the design science methodology, evaluation provides evidence that the solution artifact developed is applicable; hence, this chapter seeks to provide some evidence that the model actually works in managing the evolution of a web service in a consistent and transparent manner. This chapter discusses the results of the experimentation carried out on the proposed web services evolution model based on web service contracts.

The research described an evolution scenario in Chapter 3 where a StockQuote web service was evolved from service version 1 through to 3, and this is the context in which the results presented in this chapter were obtained. The StockQuote web service firstly started with version 1.0, where only one parameter was required to invoke the web service, evolved to version 1.1, which required two parameters and returned extra information and lastly evolved to version 1.2, which expected an invocation message with three parameters.

The results obtained are then compared with the principal goal of showing the utility of the model versus the cases in which the web services are not proxied. It should be appreciated that although there is low degradation in the Quality of Service when the web service is invoked through the proxy, the response time differences are small and can be tolerated in most non-mission critical systems. Furthermore, from a software engineering point of view, there is always a tradeoff between performance and maintainability so this result was not unexpected. It is apparent that the proxy added more programming logic to alleviate evolution challenges but in turn this had an adverse effect on performance. This is discussed further, in the experimental results section. Like

all conflicting requirements, it is not possible to achieve the best from all a balance needs to be struck.

Performance testing needs to be done as it is highly important to the success of the software systems of today. Fixing problems is costly to any organisation and thus Software Performance Engineering (SPE) tries to ensure that all software meets the expected performance objectives. Responsiveness and scalability are the dimensions that are of importance in SPE. Scalability refers to the ability of a system to meet the performance objectives as the demand for service and pressure on the system increase ("Load Testing Overview | Load Testing," n.d.), while responsiveness refers to the timeous service of the system to its requestors. SPE is a systematic quantitative approach to constructing software systems that meet performance objectives.

To measure performance for web services we need to select the metrics that are of importance to this study and which have been considered by other experts in this field. Using performance testing tools enables the collection of information about the web service and helps in decision making processes such as deciding on which web service to choose over the other on the basis of performance and not just the fees and terms of use.

6.1. Web Service-Performance testing

Web service performance testing is carried out in order to ascertain how well a system performs under a particular workload and to demonstrate that the system meets certain performance criteria. Testing also helps to identify how the system performs when exposed to specific conditions in each scenario. After testing, the stakeholders in the systems will need to know, in general, the performance metrics associated with the system, which will in turn determine the quality of service they will get from a web service, (Wala and Sharma, 2014). For example, Clients may need to

know the Response times and Throughput of an API apart from just the pricing. Service Providers need to know the resource demands of the APIs under different workloads to identify the critical use cases, to assess risk and to establish performance objectives that meet the client demands and expectations. Thus, web service performance metrics can be categorised into server-side and client-side metrics.

Server-side: can be generated by stress testing on the server end.

- Server throughput (number of requests per second)
- Latency (time between request arrival and response being served)

Client-side: distributed load testing or client-side monitoring can be used to gather this data.

- Latency (WS processing time + network latency) - from service call to earliest response bytes
- Throughput (average byte flows per minute / average transactions per minute)
- Error rate (identifies the dependability of the service)

Both server-side and client-side web service performance testing can be conducted through the use of testing tools. With the growing popularity of web services, a number of testing tools have surfaced in order to help verify that a web service does what it is supposed to do. Table 6.1 shows a summary of some of the tools that are commonly used in web service performance testing. Table 6.1 also gives a brief description of the test tools considered as at the time of doing this work. SoapUI was determined to be the most suitable testing tool for this work.

Table 6.1: Some tools for web service testing

LoadRunner	An HP-Mercury lab performance testing tool for testing Web 2.0 and mobile web applications
JMeter	An Apache performance testing tool for static and dynamic server resources
Rational Performance Studio	A proprietary IBM tool for testing web and server applications
SoapUI	The Swiss-knife web service simulation and performance testing supporting SOAP, REST, HTTP, JMS and other protocols

6.2. SoapUI

SoapUI is a cross-platform open source tool allowing for rapid creation of compliance, functional and loads tests. It provides standards support and also supports industry-leading technologies from REST and SOAP based web services to databases to JMS enterprise messaging, including scripting support for personalised automation of tests. The design of SoapUI is simplified so that anyone can have a complete testing experience using a GUI. With the GUI, any user can create a test from simple to complex scenarios ("What is SoapUI? | About SoapUI," n.d.).

6.2.1. Functional Testing

Functional testing focusses on checking the validity of a web service. To test functionality, SoapUI reduces the challenge of manually rewriting code to perform the functional tests using the Form Editor which allows a user to drag and drop controls. The functional test is generated using the

WSDL file of the service to create a TestCase. Assertions are added in the TestCase in order to validate the validity of the result in the response message for correctness. An example of an Assertion can be a Response SLA, in which a response time to be validated can be set to 500ms. That is to say, if the response is returned from the web service in a time less than or equal to 500ms, then the assertion is a valid assertion and the test passes, else the assertion validates to a failed test. Figure 6.1 shows a functional test with three assertions. The first assertion is to check if the service is giving back a valid SOAP response after having been invoked by a SOAP request based on the contract requirements from the consumer. The result of this assertion shows that the service responded with a valid SOAP response. The second assertion checks the compliance of the response from the StockQuote service to see if it complies with the schema / condition set in the contract. The result in the example in Figure 6.1 shows that the response from the web service is schema compliant. The third and last assertion was set to test if the SOAP response actually contains the information that the consumer will be expecting to receive through the provisions of the contract. This last assertion also shows that the web service was successfully invoked and accepted the incoming request or not. The Not SOAP Fault assertion is valid, showing that the web service has responded with a valid SOAP message that contains the data as promised by the provider through the contract. The functional test was therefore successful.

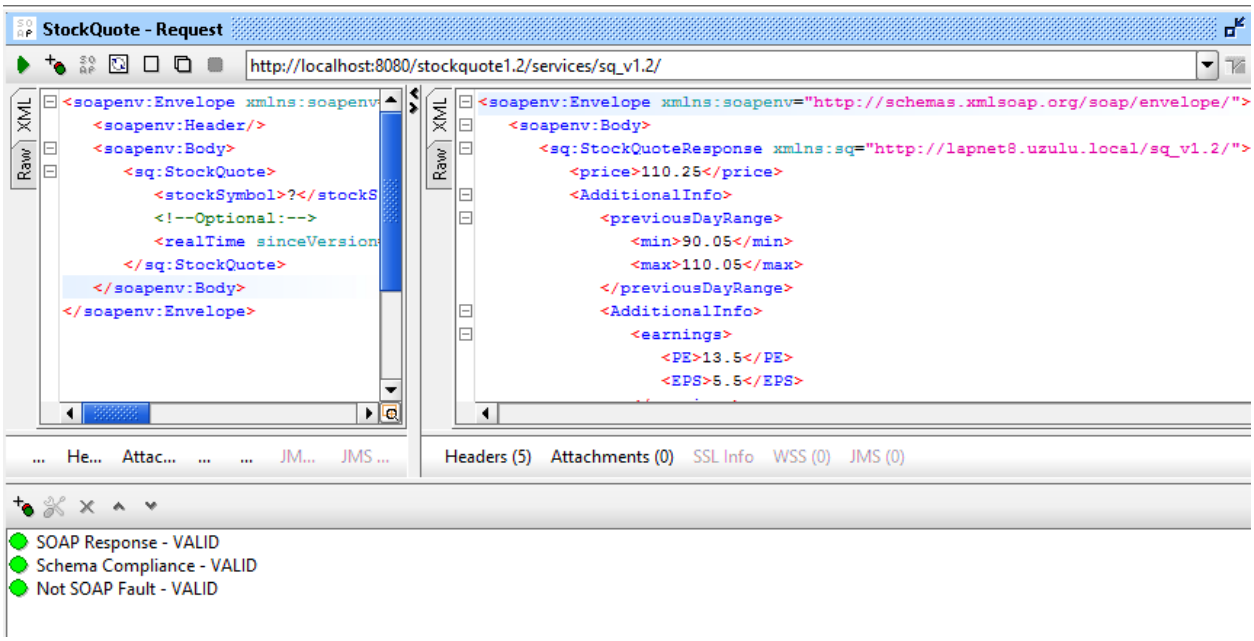

Figure 6.1: Valid positive functional test

Figure 6.2 shows an unsuccessful functional test. In this functional test, the schema that was sent by the consumer (SoapUI), may appear to be a valid SOAP request, however, a request was generated that was not compliant with the expectations of the StockQuote web service, and it was that which was used to invoke the web service. The web service responded with a valid SOAP response, which was compliant with the expected schema but the last assertion resulted as a failed assertion. The assertion expects to receive a response which is not a fault message. The web service response is a fault message because it has generated an error upon picking up that the web service request message was not a valid request according to the binding contract.

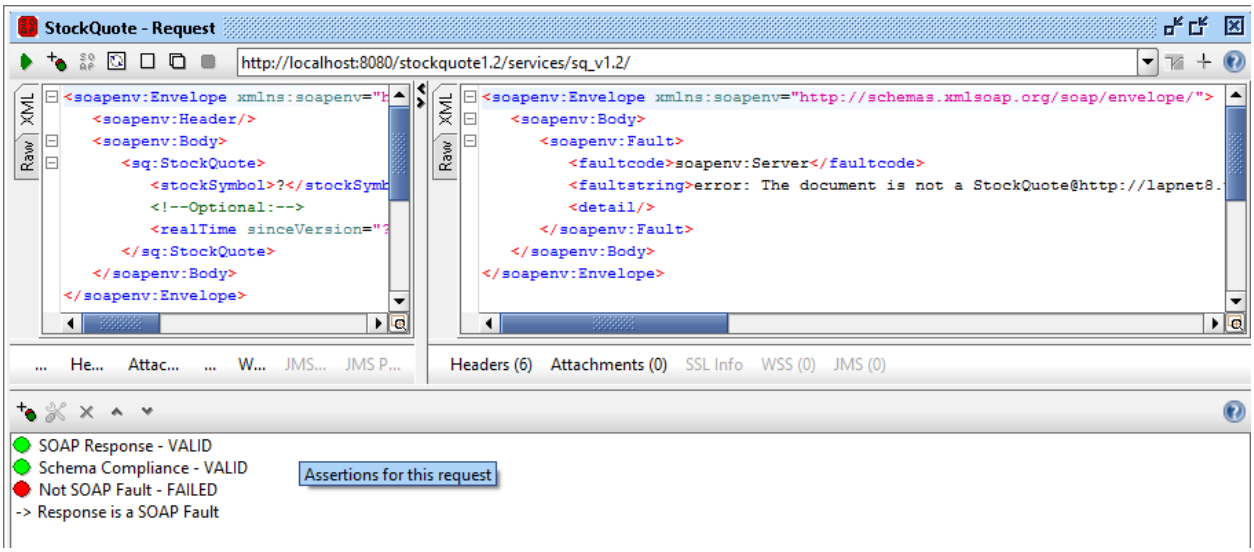

Figure 6.2: Valid negative functional test

6.2.2. Load Testing

As a web service gains popularity and more users opt for it compared to other services out there, the load on the web service and consequently on the host web service server machines increases. It therefore becomes imperative that load testing be conducted to know the carrying capacity of the web service. Load testing is performed by increasing the load on the web service to observe how the system behaves as the load increases. During the load test, performance metrics like response times, server memory usage and web service throughput can be measured.

6.2.3. Average response times

The response time tells a story of how long it takes after a request is submitted for it to be processed and for an appropriate response to be received by the web service requestor. The average response time is therefore calculated from the collection of response times and subsequently mathematically dividing the total time by the number of times the time has been collected/added. This average response time is not a functional requirement on a service but is classified under non-functional

requirements and is used as a quality of service metric. The industry standard of acceptable response time for a fast service is 200ms (Levy, 2014).

6.2.4. Scalability

Scalability Testing is very much like Load Testing. Scalability Testing is more about testing the elasticity of the web service to see how much the web service can accommodate varying volumes of traffic. For a scalability test, the size of a request or the complexity thereof may be increased, as opposed to increasing the number of requests. This may involve sending nestled requests, larger attachments, or larger requests.

6.3. Experimental Results and Analysis

6.3.1. Average response time

Using the original version of the StockQuote (StockQuote 1.0), the TestCase was repeated to establish a base case. The experiment was repeated up to 50 times and the response times, measured in milliseconds, were recorded in Table 6.2. The experimentation was repeated in order to minimise the effects of possible anomalies that may occur in the system during experimentation. Anomalies such as a spike in memory usage by other running programs may cause slower response times in the experimental results, hence the repetition helps in establishing a more reliable average response time.

Table 6.2 shows the results of a TestCase for StockQuote version 1.0 which were obtained when a service request was issued to StockQuote 1.0. The response time is the time it took for the service to process the service request and send the appropriate response upon receiving a service request, in this case **without** the proxy implementation.

Table 6.2: Response times for StockQuote 1.0 TestCase

18	15	13	10	11	9	11	14	11	12
10	12	11	12	18	11	10	10	11	11
11	15	11	10	9	12	12	12	10	11
9	10	10	13	11	13	14	12	10	11
9	13	11	9	10	12	13	10	10	11

The response time is not the same in all instances because the host system is also processing other jobs outside of just running/hosting the StockQuote web service, which then makes it ideal for us to repeat the experiment to compute an average response time that will serve as the base response time for the web services on the particular host.

Average Response time = Total of the recorded times / Number of times observed

Average Response Time = 574/50

= 11.48ms

The Average Response Time rounded off to the nearest whole number becomes 11ms, which is the Base Response Time that was used as the benchmark for the other web service version tests. Given that the average response times are all well below 200ms, which is the industry benchmark for a fast service (Levy, 2014), this research concludes that the proxy model does not degrade the quality of service in terms of response time.

During the establishment of the base response time, a snapshot of the TestCase of the service transaction was captured and is presented in Figure 6.3. Figure 6.3 shows the base TestCase where the SoapUI test suite was used to test the web service StockQuote v1.0 and to collect all the

necessary information. In the TestCase presented in Figure 6.3, we have labelled items from 1 through to 7 that we wish to draw the reader's attention to.

1) Item 1 is the local address of the host of the StockQuote web service which is defined in the web service contract file. http://localhost:8080/StockQuote1.0/services/sq_v1.0/ also describes the endpoint of the web service
2) Item 2 is the SOAP service request message that is sent to invoke the StockQuote v1.0 web service. It contains the one parameter that is required by the web service in order to invoke the web service, the <stockSymbol>
3) Upon receiving the SOAP request, the web service validates the request based on the restriction imposed in the web service contract, and if valid, is passed onto the web service for processing. The web service returns the appropriate SOAP response containing the <price> as per the StockQuote of the day. This is received and displayed by the client, in this case SoapUI
4) In this TestCase, the correct parameters were submitted to the StockQuote v1.0 web service, evaluated to match the conditions specified in the web service contract and the appropriate SOAP response was successfully sent back to the client, hence the TestCase passed and SoapUI generated no errors and hence the green colour against the TestCase to imply success
5) Item 5 shows the number of assertions that are tested during the web service execution. In this TestCase, the StockQuote return a valid SOAP response, whose Schema is valid according to the contract that is in effect between the consumer and the provider. Since the web service functioned correctly and did not return a fault message, the assertion to test for faults returns a valid "not SOAP Fault" status

6) SOAP Request and Response pairs have a processing time between them. The time it takes from when the web service receives a SOAP response to the time it sends back the appropriate SOAP response is a measure of the response time of the web service

7) The SOAP response in Item 3 has a size, and this size is the total file size that is sent across the wire from the web service to the client/consumer. Therefore in this instance it takes an average of 11ms for the StockQuote web service to respond to a SOAP Request

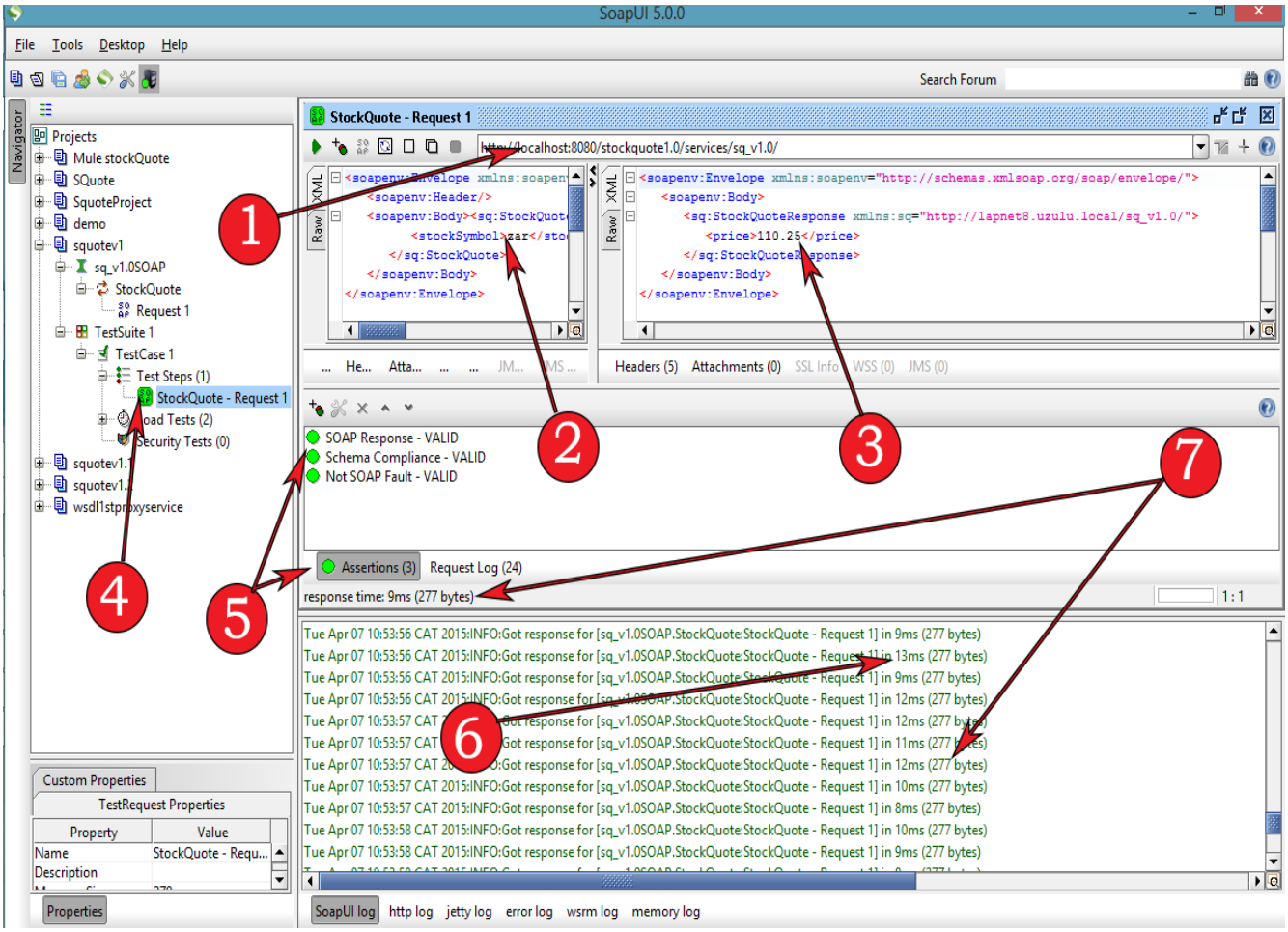

Figure 6.3: Test case for StockQuotev1.0

6.3.2. Load Tests

As noted earlier in Section 6.2, the purpose for executing load tests on the StockQuote web service is to monitor how the web service behaves under an increasing load. The load tests will show the effect of including the proxy as proposed in the model for web service evolution management and help establish if it is a disservice to have the proxy in place.

There is no prescribed standard for the load testing of web services. As a result, experiments were carried out to establish the test ranges and test scales that were to be used as the basis for the load tests that were carried out on the model. Figure 6.4 shows a load test being performed on the StockQuote version 1.0 web service, and we have labelled the items that we describe as 1 to 6 below.

1) When a web service is published, a consumer is constructed to utilise the web service. In binding to the web service the consumer creates what is represented in SoapUI as a "thread". The thread generates the requests/calls to the web service, but this should not be mistaken to imply that a thread is equivalent to one consumer. The thread can represent any number of consumers depending on the nature of the web service and the request criteria being employed
2) SoapUI can mock different strategies of performing load testing. For the purposes of this work, the simple strategy was employed throughout in an effort to remain as close to a real-life-scenario as possible
3) The test delay control regulates how much time a request should wait after a previous request. In the experimental part of this work, the test delay was set to 0, to allow for complete random testing without any artificially induced wait periods

4) Item 4, the Random control, regulates how randomly the thread can send a web service request/test. The random value ranges from 0 to 1 where 1 signifies a degree of complete randomness

5) Item 5 is the limit to how long the load test should run. After repeatedly experimenting to find what amount of time would be significant for this research, it was determined to keep the Limit of each test to a maximum of 60 seconds. 60 seconds in theory, implies that a single thread will generate up to 60 executions of the TestCase. assuming that the StockQuote service is only requested once by any consumer, this would theoretically imply that the web service is serving up to 60 consumers per minute

6) Item 6 in Figure 6.4 shows how much memory has been dynamically allocated to executing the test cases and how much is actually in use when the test case is in execution

Figure 6.4: Load testing for each service version.

Figure 6.5 shows the execution of the load tests for all the evolved StockQuote versions as they are run simultaneously with the aid of the proxy. Without the proxy this would not have been a possibility as all the StockQuote service versions would have been independent and thus could not have yielded meaningful results as they would have even possibly been implemented on different hosts with different specifics.

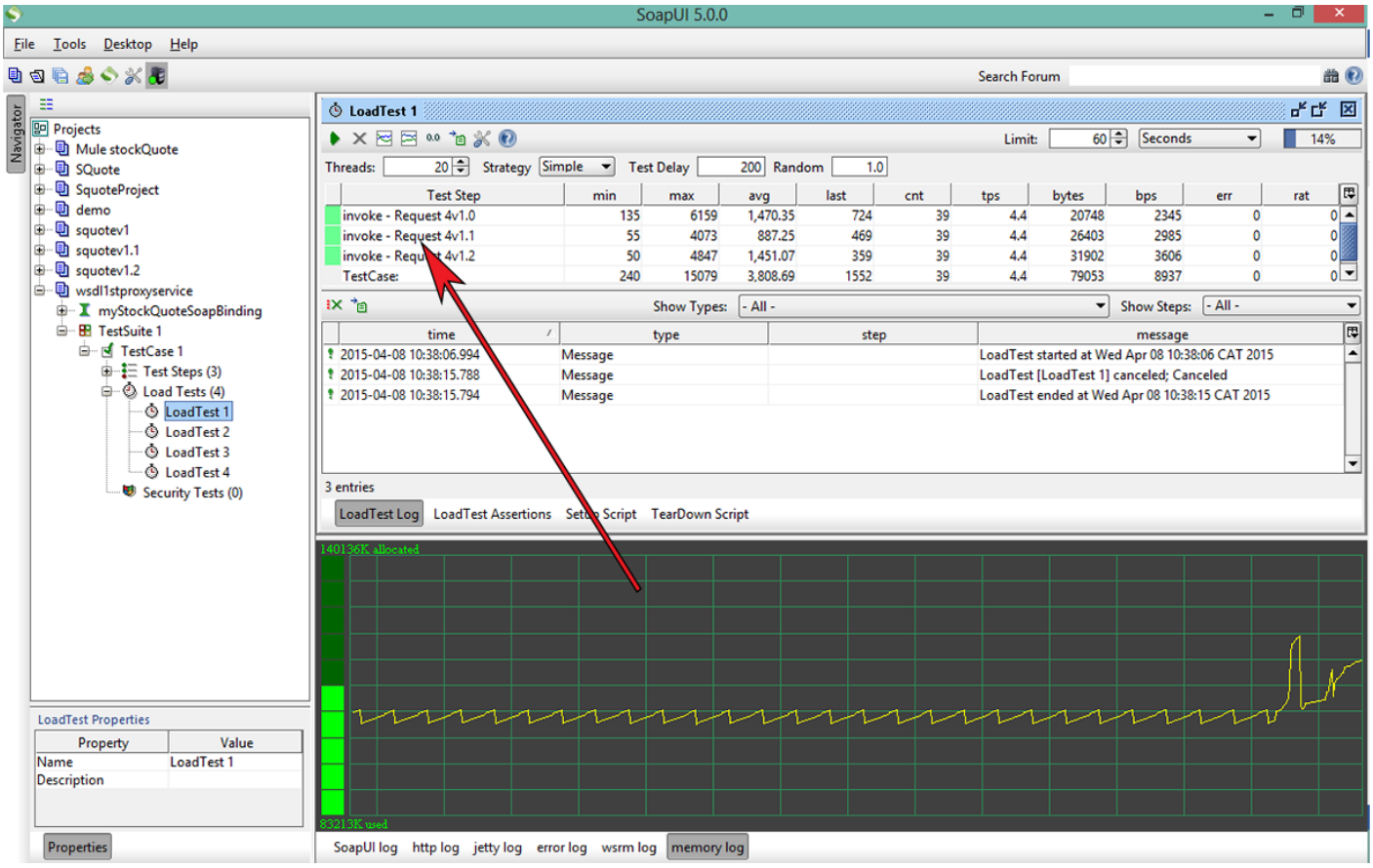

Figure 6.5: Load testing for all services through the Proxy.

6.3.3. The Average Throughput

After performing a series of workload tests on service version 1.0, both with and without the proxy, the data gathered was used to design the test-cases and the workloads to apply during the tests. This was done in order to determine the characteristics and specifications that would be needed to

configure the workloads in order to collect the data across all versions. No historical data was found on how to design these performance tests hence the design was based on the initial experimentation in the current custom test environment as suggested by Weyuker and Vokolos (2000). The goal was to determine the scales by which the workload would be increased to determine whether there was likely to be a hardware performance limitation or that the system or setup would fail while the load was increased. This was a "*stress-test"* test performance in an attempt to determine the elasticity and the breaking point of the setup. The workload in this case was the number of requests generated by the threads (Figure 6.4: item 1). Figure 6.6 shows the relation between the number of threads and the virtual users.

Given that the StockQuote web service, implemented here as StockQuote version 1.0 through to version 1.2, is invoked only once per transaction by exactly 1 user, then each thread is a group of users generating one request and receiving their response in an estimate time of 11ms. The graph in Figure 6.6 approximates a relation in which the number of requests is directly proportional to the group of users (threads), meaning that each thread contains roughly 500 users each invoking the service only once at random time per minute. So the StockQuote web service is expected to service up to and beyond 20x500 requests per minute in the implementation where an ESB and a proxy have not been included. The throughput in SoapUI is calculated as transactions per second (TPS), which in the experimentation was calculated on the basis of the actual time passed and was obtained using the simple formula

TPS = Count (cnt)/ Seconds passed.

That is, a test-case that runs for 60 seconds and having handled 510 requests will have a TPS of 8.5.

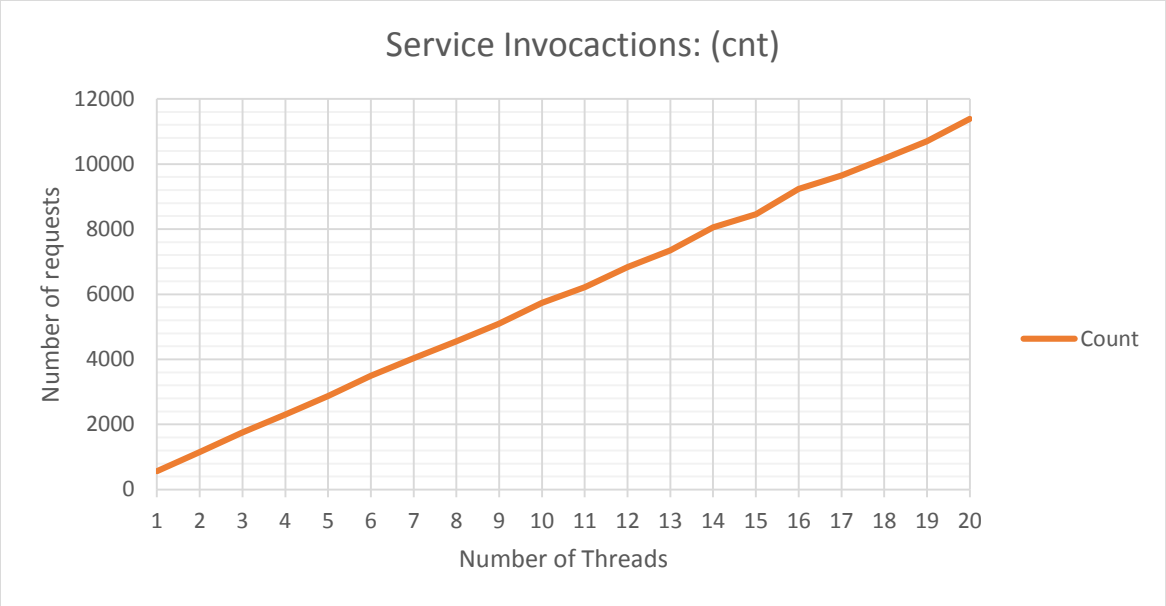

Figure 6.6: Relation between the thread-count and the number of requests.

Figure 6.7, Figure 6.8, and Figure 6.9 show the throughput of the StockQuote service versions 1.0, 1.1, and 1.2 respectively as the workload was increased, in what was designed as the first set of tests. The throughput was observed to be higher across all service versions in the case where the workload was applied directly between the virtual clients as simulated through SoapUI and the service version implementation in the eclipse experimental environment. The setup as depicted in Figure 4.2 was taken as the ideal setup where the service client and the service provider are both residing on the same machine. Hence, there are no network delays experienced, and no domain name lookup and host resolutions required.

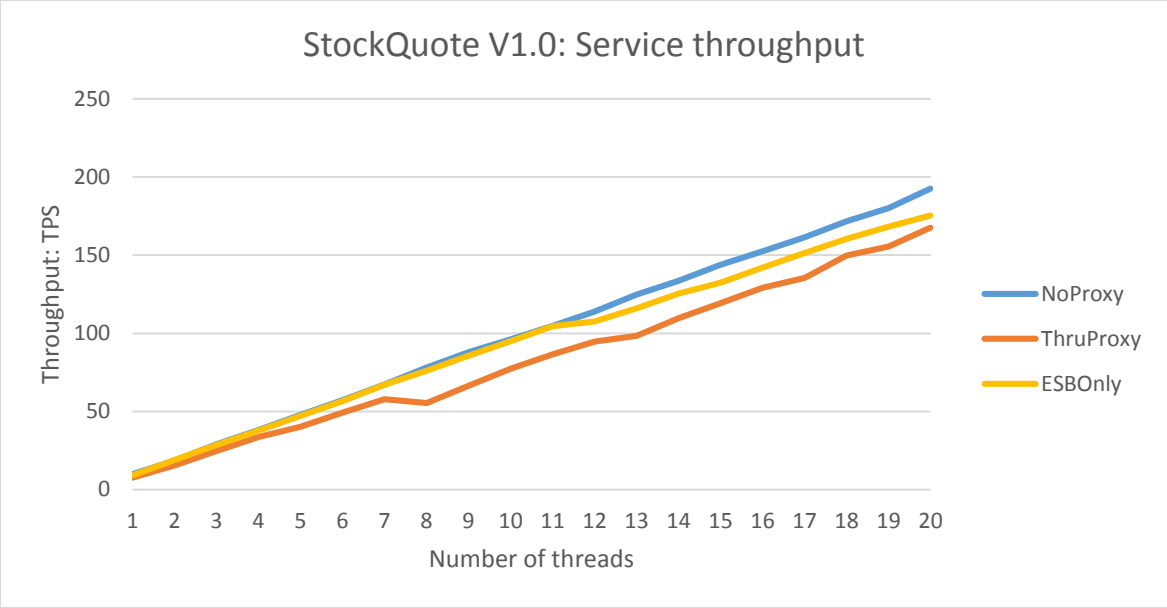

Figure 6.7: Throughput for service version 1.0

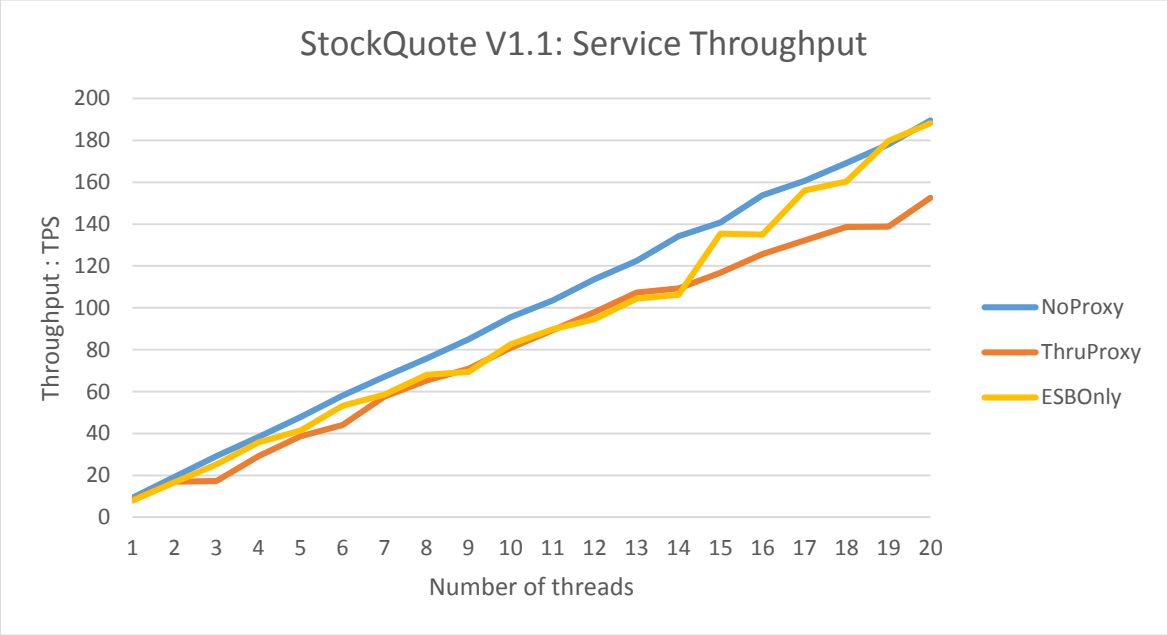

Figure 6.8: Throughput for service version 1.1

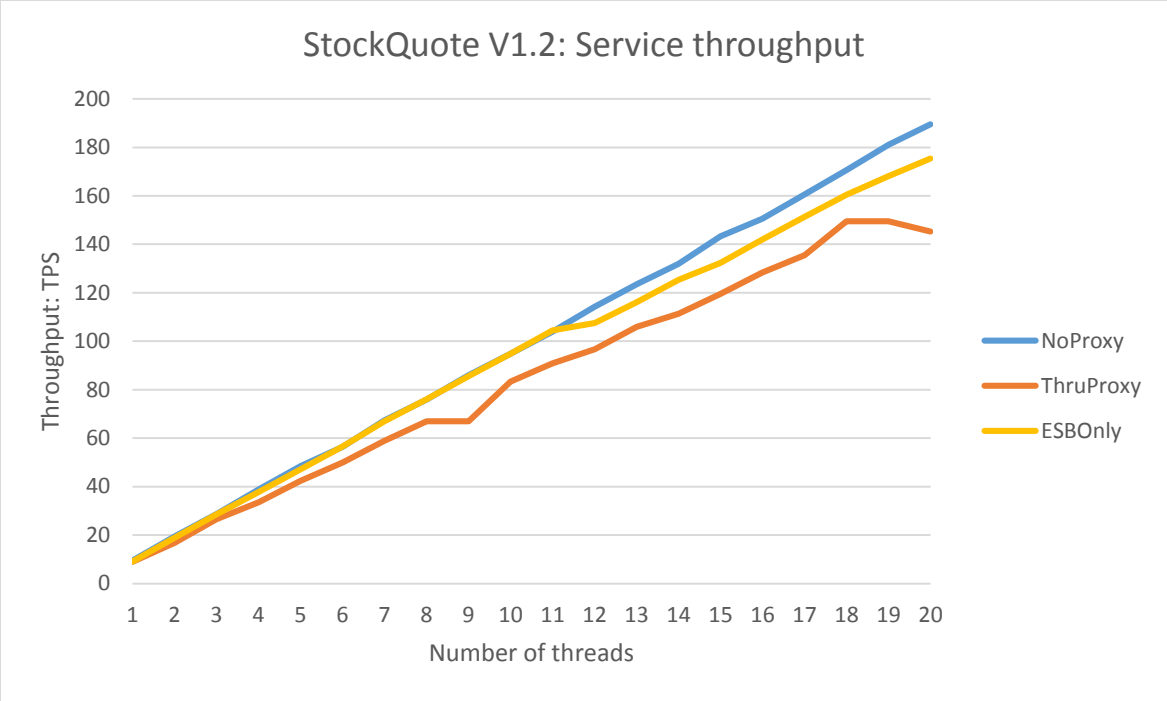

Figure 6.9: Throughput for service version 1.2

In the second set of tests, the service client and the service provider are implemented on different hosts, unlike in the direct request of the experimental setup of Figure 4.2. Like the setup in Figure 4.3, an ESB was implemented in order to try and mimic as closely as possible a real-world setup. Across all service version tests there is a drop in the throughput as the workload increases. The drop or loss in throughput is expected and inevitable due to the processing of requests as they pass through the ESB. The ESB receives the service request on one port, has to read it and interpret the destination of the request and route it to the appropriate endpoint. Likewise, the service response goes through similar processing and is routed in the ESB back to the requesting service client. The ESB processing was treated as a black-box because this research is not concerned about the details of this processing as the ESB forms part of the architecture in a real SOA implementation.

The third test set was performed on the solution model implementation as demonstrated in Figure 4.4 and Figure 5.1. A further loss in throughput was experienced as, inevitably, more processing

overheads are encountered since not only does the ESB perform its default routing functions but the implementation of the service transformation proxy relying on the service contract to know what to transform also adds to its processing time. The proxy receives an incoming request, checks which contract that request matches, compares with the current implemented service version, transforms the request to comply with the expected request for the implemented service version (typically the latest version), sends the request to the service endpoint, and waits for a response. Upon receiving the response the proxy transforms the response to match the contract version that had been identified as the requesting client implementation version, and routes it appropriately to the calling client. This contributes to the delays in processing of requests and results in a lower throughput than in the ideal scenario depicted by Figure 4.2.

Another factor affecting the throughput is the response time, and the response time is affected by the file size of the SOAP request and response being transmitted. The SOAP request/response in the StockQuote example increases in size as the extensions to support evolution from version 1.0 through 1.1 to 1.2 are implemented as described in the Dataset Amendment pattern for evolving web services (Daigneau, 2011). So, throughput loss is due not only to the processing overheads introduced by the proxy but also to the size of the resultant SOAP messages being exchanged by the systems in question.

It was observed that there is a loss in throughput due to the introduction of the proxy as proposed in the model. Hence, understanding to what extent this loss in throughput actually affects productivity is required, as is whether or not it is significant enough to conclude that the introduction of the proxy is a disservice or a benefit to productivity. Take, as an example, a taxation scenario: it would be unfair to state that a fixed-tax of R500 will be charged on everyone's salary. Having the fixed amount would mean that a person being paid a salary of say R2000 will be

charged R500 tax and a person receiving a salary of R10000 will also receive a tax of R500. What would however be considered a fair taxation setup is where a percentage of tax is charged on the income, say for instance a tax of 15%. This would mean that everyone pays 15% of what they get, as opposed to some being overtaxed and others being undertaxed in a fixed-amount tax scenario. Similarly, this work expresses the throughput loss as a percentage of the expected throughput in order to understand the impact the proxy has. This is expressed using Equation 6.1:

$$\text{Percentage loss} = ((A-B)/A)*100$$

Equation 6.1: Calculating percentage loss in throughput

Where A = the expected number of requests, B = actual number of requests.

Percentage loss in the first case where requests and responses pass through the transport media only, i.e. through the network and ESB without a proxy implementation is calculated as: Percentage loss1 = ((Num. requests NoProxy – Num. Requests ESB) / Num. requests NoProxy)*100

= 3.48%

Percentage loss in the second case where the proxy was implemented is shown as:

Percentage loss2 = ((Num. requests NoProxy – Num. Requests Proxy) / Num. requests NoProxy)*100

= 13.84%

The effective throughput loss due to the proxy is expressed and the difference between the throughput loss in the first case without a proxy implementation and the second case with the proxy running was expressed using Equation 6.2.

Effective loss = loss2 –loss1

Equation 6.2: Calculating effective loss in throughput.

The result of the effective loss from this equation is equal to 10.36%. Therefore, the proxy used here in this work as an instantiation of this model causes roughly a 10% loss in throughput as some requests are queued up waiting for service. It is at this juncture that one may raise the question, "Is 10% not too much of a loss?" To answer this question, let us firstly look at the goal of this work which was to:

- Formulate a contract-based model for managing evolution of shared services in order to minimise disruptions to service consumers when a service changes.

The priority of evolution through Amendment of the Data Structures of a web service is minimising the probability of breaking consumer support. The goal of this work is to maintain web service functionality and web service evolution transparency to the consumers through the implementation of a contracts-based proxy while the service undergoes continual change and improvement to meet changing consumer and platform requirements. The proxy in the model described in this work maintains but is not limited to the operation of at least three web service versions concurrently, with the only downtime expected being maintenance-time or upgrade-time. The proxy eliminates the need for more resources that would otherwise be required to maintain the same three web service versions running in separate instances at the same time; the proxy can be regarded as a *"1 for 3 or more solution"*.

Secondly, judging from Figure 6.7, Figure 6.8 and Figure 6.9, while there is an evident drop in the number of transactions serviced as the load is increased the difference in the trends of the graphs is small. For fewer requests the throughput is almost the same for the ideal situation where the proxy is not implemented versus when the proxy in implemented, and that loss only increases to a

calculated maximum of roughly 10% when the system is under a heavy workload. This research argues however that, in reality, the requests will not be at peak traffic continually but will subside and vary depending on consumer usage and times, even that latency will cause requests to arrive on the service at different times as opposed to the simulation in this work. Hence, the loss in throughput will generally be less than the maximum calculated. This work reiterates at this point that the throughput loss does not mean loss of service but just a slower response time from the web service while it processes queued up requests. Consequently, this research concludes that the benefits associated with the implementation of this contracts-based proxy outweigh the possible throughput loss.

6.4. Economic and Industrial Implications

A discussion of the web service performance metrics cannot be conclusive and complete without us visiting software economics. Software cost estimation is a large topic, the discussion of which in this section alone would not do justice. This section will present an overview and tie in software cost estimation with the benefits associated with the contracts-based proxy model for managing evolution of web services. Software costs may include but are not limited to travel and training costs, hardware and software costs. Software productivity is viewed as the rate at which the software is produced by the developers, including the documentation of that software. The number of lines of code can be used as one of the bases for the measurement of software productivity. The other measure that can be used is the function points of the software being developed. However, there are challenges, such as how does one estimate the function points of software yet to be developed and how does one estimate the total time that will be required for the project? Lines of code, as a measurement, was first proposed when programs were typed as instructions on cards where each card had one line of code. However, today's programming languages allow for multiple

statements to be written in one line or spread across several lines. A line of code nowadays refers to a line of code that is hand-written, not machine generated, and also not a blank line. The line of code is referred to as Source Lines of Code (SLOC). KSLOC is derived from SLOC where the K stands for kilo (a factor of 1000, i.e., SLOCx1000), for example, 8 KSLOC = 8x1000 SLOC therefore, is equal to 8000SLOC.

There are several software cost estimation techniques and these include but are not limited to: The Algorithmic Model, Expert Judgement, Analogy, and Parkinson's Principle (Boehm, 1981; Bryant and Kirkham, 1983). Table 6.3 presents a summary of some of the cost estimation techniques.

Table 6.3: Summary of cost estimation techniques (Sommerville, 1982)

Estimation technique	Description
Algorithmic cost modelling	A model based on historical cost information that relates some software metric (usually its size) to the project cost. An estimate is made of that metric and the model predicts the effort required.
Expert judgement	Several experts on the proposed software development techniques and the application domain are consulted. They each estimate the project cost. Their estimates are compared and discussed. The estimation process iterates until an agreed estimate is reached.
Estimation by analogy	This technique is applicable when other projects in the same application domain have been completed. The cost of a new project is estimated by analogy with these completed projects.

Parkinson's law	Parkinson's Law states that work expands to fill the time available. The cost is determined by available resources rather than by objective assessment. If the software has to be delivered in 12 months and 5 people are available, the effort required is estimated to be 60 person-months.
Pricing to win	The software cost is estimated to be whatever the customer has available to spend on the project. The estimated effort depends on the customer's budget and not on the software functionality.

Each estimation method has its strengths and weaknesses and Table 6.4 summarises these.

Table 6.4: Strengths and weaknesses of software cost-estimation methods. (Boehm, 1981)

Method	Strengths	Weaknesses
Algorithmic model	• Objective, repeatable, analysable formula • Efficient, good for sensitivity analysis • Objectivity calibrated to experience	• Subjective inputs • Assessment of exceptional circumstances • Calibrated to past, not future
Expert judgment	• Assessment of representativeness, interactions, exceptional circumstances	• No better than participants • Biases, incomplete recall
Analogy	• Based on representative experience	• Representativeness of experience
Parkinson	• Correlates with some experience	• Reinforces poor practice
Price to win	• Often gets the contract	• Generally produces large overruns

The COnstractive Cost Model (COCOMO) is perhaps the most well-documented and well-known software effort estimation method (Roetzheim, 2000). COCOMO popularised the SLOC as an estimation metric and is used to calculate the Effort for a software project in months. The simplest relationship between the Effort and the SLOC is expressed as the product of the productivity and the KSLOC. Over the years researchers have found some common values for the Productivity and Table 6.5 provides a list of these.

Table 6.5: Linear productivity factors for software development (Cost Expert group, www)

Project Type	Linear Productivity Factor
COCOMO II Default	3.13
Embedded development	3.60
E-Commerce development	3.08
Web development	2.51
Military development	3.97

With the KSLOC and the productivity factors in Table 6.5, an example of how the Effort would be derived is given by Equation 6.3:

$$\text{Effort} = \text{Productivity} * \text{KSLOC}$$

Equation 6.3: Calculating effort in man-months (MM) (Cost Expert group, www)

Hence: Effort = 3.13 * 8 = 25.04 Man-Months, where 3.13 is the Linear productivity factor obtained from Table 6.5 and 8 “Kilo” is the example number of lines of code.

Productivity does vary with project size and how verbose or concise the developers are. Hence, an exponential penalty is introduced to penalise large projects for reduced productivity. However, the calculation of Effort on the basis of Equation 6.3 will suffice for small projects (Roetzheim, 2000).

In the development of the StockQuote web service the developer is expected to write the contract and the source code for the implementation of the StockQuote service. The estimates of the amount of time for the development of the StockQuote web service versions 1.0 through to 1.2 are given in

Table 6.6. This work classified the web service as development of an E-Commerce platform, hence the use of the 3.08 value of the Linear Productivity Factor in this work's estimations. The estimated Effort in

Table 6.6, working with only the human hand-written code, was obtained using Equation 6.3.

Table 6.6: Estimate effort for the development of the StockQuote web service and the proxy

Description	**Contract SLOC**	**Impl. SLOC**	**Total SLOC**	**Effort @ LPF of 3.08**	**No. of days (Effort)**
WS-StockQuote v1.0	63	45	0.108	0.33	10
WS-StockQuote v1.1	100	61	0.161	0.50	15
WS-StockQuote v1.2	119	66	0.185	0.57	17
Contracts-based Proxy	52	62	0.114	0.35	10

Now, what does this all mean when the contracts-based proxy web service has been implemented? At the beginning, development and implementation of the proxy will take approximately 10 days

plus the implementation of version 1.0 of the StockQuote, which is approximately 4 days (considering only the 0.045 KSLOC for the version 1.0 implementation). This comes down to an estimate total of 14 days development time at the beginning of the StockQuote web service's lifetime.

At the next stage, the StockQuote web service is evolved to version 1.1. At this stage, only an update to the proxy implementation is observed while the contract is maintained to support the version 1.0 consumers. The additional lines of code for the update to the proxy implementation were 13 lines of code in upgrading the proxy implementation without repeating code. This translates to an estimated 3 hours of development time as opposed to implementation of a complete new StockQuote version 1.1, which would take an estimated 10 days according to the results in

Table 6.6. Similarly, at the last stage of the StockQuote the web service was upgraded to version 1.2. The proxy updates required an additional 27 lines of code in the implementation of the proxy in order to support all StockQuote web service versions 1.0 through to 1.2 concurrently. This upgrade comes at an estimated two days development and implementation cost, as opposed to the tabulated 17 days to implement a completely new StockQuote version 1.2 web service.

Take for example, the VirtualBox API. The VirtualBox API is a SOAP based API allowing deployment of virtual machines, whose implementation was found to be an approximate 54 KSLOC and has had up to 10 major and minor version releases which ALL contain breaking changes (Espinha *et al.*, 2015). All backward compatibility for clients was disregarded and thus costing development time, effort and unplanned expenses to developers in re-implementing systems that were relying on VirtualBox; like the phpVirtualBox client. VirtualBox is a SOAP based API thus has only a single point of integration with its clients (the technical Contract), however this means that clients are tightly coupled to that Contract. Should the provider issue a

breaking change, it is inevitable that all phpVirtualBox clients will fail. This can be remedied by the implementation of the solution proposed in this work, the contracts-aware proxy to manage that backward compatibility allowing for client developers to gracefully evolve their systems over a period of time.

Working with the assumption that each web service is hosted on a single machine, implementation of the StockQuote web service plus the proxy to manage the web service and its evolution would require only two host machines - one host for the proxy and the other for the web service implementation. However, without the proxy implementation, each web service version will be hosted on its own host machine. This would prove uneconomical in terms of the resources that a service provider would need. On one hand, as more versions are introduced more host machines would need to be provided. On the other hand, the contracts-based proxy approach maintains two host machines as new versions are introduced. The maintenance, not only of the web service evolution processes, but even of the infrastructure required when the contracts-based proxy approach is used, becomes easier and cost effective to the service provider. Managing as little as two host machines is easier compared to maintaining multiple concurrently running versions of the web service, multiple software environments, multiple operating systems and multiple hardware platforms hosting the services.

It is of key importance to note that the application of the contracts-based proxy model results in the ability of a service provider to service both old and new consumers concurrently with no need for additional hardware and most importantly without requiring any consumer to forcibly upgrade to the new service version. With the implementation of the contracts-based proxy there is only minimal disturbance to the already existing consumers when the system is administratively taken down for scheduled maintenance and updates.

6.5. Summary

This chapter discussed the evaluations performed on the proposed model through the implementation of a contracts-based proxy that was applied to a StockQuote web service and its derivatives in order to investigate the model's validity and applicability in a real world scenario. In the next chapter, the conclusion and future work is presented. The success of the model was demonstrated by the fact that this work implemented a proxy that was able to handle the incoming requests from different versions of consumers and successfully serviced the requests consistently. With the aid of a standard and common tool for testing web services, performance and compliance tests were conducted on the model.

Web service development comes at a cost. It was found that there was extra processing time required in the proxy, as would be anticipated in any case where additional processing is implemented between a consumer and a service being provided. The increased processing time results in lowered throughput of the service. In this implementation of the model, there is a small drop of only 10% in the achieved average throughput. Although there is this estimated drop in throughput, the main advantage of the model presented in this work is that the proxy would be handling all incoming requests for all the running StockQuote versions simultaneously, without service disruptions to consumers.

The minimal implementation of the contract-based proxy model, having taken the assumption of one host per web service into consideration, would entail maintaining only two hosts while the web service is evolving during its lifespan. Maintaining service without the contracts-based proxy would imply that the number of hosts required increases with each new version that is introduced while trying to keep the old consumers serviced. This puts a financial strain on the service provider

in purchasing the required hosts and the technical administrative labor costs for maintaining the infrastructure.

Using the COCOMO software estimation technique for small to medium projects, it was established that it takes less time to upgrade the contracts-based proxy in order to support a new web service version than to implement a new version while maintaining the old host to support older consumers. This makes the implementation of a contracts-based proxy cost effective on the part of the service providers and consumer friendly to the customers as it does not force the customers to upgrade the consumers.

CHAPTER SEVEN

7. CONCLUSION AND FUTURE WORK

7.1. Summary of the research

Web services will always change to meet varying change-requests from all stakeholders in these services. While SOC has increased the flexibility and agility of organisations in adapting to an ever changing business operating environment and enabled easy composability of services to achieve business tasks, it has also brought challenges in controlling the changes needed and changes introduced by other parties in this shared environment. Changes or upgrades may have unforeseen minor to devastating impacts on other unsuspecting consumers and may cause business disruptions, costing large amounts of revenue to organisations dependent on these web services. In order to mitigate the impact of these changes it is imperative that the evolution of web services be undertaken in a controlled manner.

The goal of this work was to formulate a model for managing the evolution of shared services in order to minimise disruptions to service consumers when a service changes. Hence, this work presented a contracts-based proxy model for managing the evolution of web services. The model presented in this work establishes control over the evolution of a web service on the basis of two main aspects, the best practice of the design of web services by contract with evolution in mind, and the introduction of a proxy relying on the designed contracts. This work, following the design science research methodology, demonstrated the efficacy and the applicability of the model using both a running scenario and consultations with professionals in the field of web services management. This work followed the design science research methodology in order to come up with sound findings. Hevner *et al.,* (2004) provided guidelines for performing a design science

research which were followed throughout the course of this work and these are described in brief below:

- **Design as an artifact:** this work produced a model which is the artifact that was used as the basis for all investigations. The instantiation of the model is also another artifact that was realised through this work. From this artifact, empirical data was collected and used for analysis
- **Problem relevance:** this work has shown starting from chapter 1 that there are still challenges in managing consistent and non-disruptive web service evolution. This work focused on addressing SOAP based web service evolution in SOC because SOAP is still dominant in commercial implementations of web services. Thus, the solution presented in this work will contribute in alleviating the evolution problem in the web services industry
- **Design evaluation:** to demonstrate the applicability and utility of the proposed model, a prototype was developed as one form of evaluation. Various experiments were then carried out using this prototype to evaluate its efficacy
- **Research contributions:** According to Hevner *et al.,*(2004), the contribution of design science research is design science knowledge. This knowledge can be in the form of artifacts such as, constructs, models, frameworks, methods, design theories and instantiations. This work produced a model and an instantiation which will enable graceful evolution of web services. Thus benefitting developers while minimising inconveniences to both service providers and service customers
- **Research rigor:** the model produced in this work was as a result of logical deductions from sound literature sources. This model was instantiated into a prototype and various kinds of assessments were then carried out to ensure the quality of the artifacts. Feedback

was also sort through conference publications and presentations. All this helped to shape the model to be what it is now

- **Design as a search process:** the model presented in this work was developed and refined through an iterative process of implementation, consultation and revisiting literature to verify its applicability in an ever-changing web service environment
- **Communication of research:** This guideline relates to how the research is communicated or shared with others. To share the work with others, this work resulted in 2 IEEE conference proceedings. Another manuscript is expected from this work and will be published in an academic journal for further investigation. The dissertation itself will be made publicly available through the University of Zululand library

7.2. Research Questions Review

In the introductory section of this work, the main research question was defined, around which the rest of the work was centered. *How can service contracts be used to incorporate and manage service evolution?* This was further broken down into four sub-questions. This work provides the responses to the research question and sub-questions.

Research sub-question 1: What is the state of the art of service evolution in SOC and what other fields can we take lessons from?

In Chapter 2, Section 2.5, the state of the art in web service evolution was presented in which this research discovered that the trends in computing are now in Services and Cloud computing. It was also noted that less of the research effort over the years has been concerned with the management and control of the evolution of web services. The first research objective was achieved in determining, through literature, how the WSDL file can be leveraged upon in managing the

evolution of a web service. More research effort is needed in addressing service evolution challenges that have been highlighted since the beginning of SOC. Lessons can be drawn from other fields, for instance, Information System, where there are mature works on evolution management, evolution life-cycles and Laws governing software management. From an engineering perspective, processes to support the evolution of service oriented systems are a challenge. Conventional development methodologies such as Object Oriented Analysis and Design (OOAD), Component-Based Development (CBD) and business process modelling, notwithstanding their usefulness, do not address the key elements of SOC. It was discovered that one key component which can be used in controlling web service evolution is the contract, as it is the one attribute that contains all the necessary information a consumer needs to know. As services evolve to meet new knowledge acquired in their domain (the context in which they are used), they also need to be maintained such that their previous state of consistency is restored but with the necessary additional essential features. Service evolution becomes a critical issue because even the smallest of changes, if incompatible, can affect a huge number of clients and consumer applications. At the same time, customers do not necessarily need totally new systems to address their new set of requirements, but they do need the familiar environment that they are already used to, but with additional features or functionality.

Research sub-question 2: How can we design a model to ensure that there will be no major disruptions to business functions after a service is upgraded or changed?

The state of the art-analysis was instrumental in helping us to learn more about the problem area of web service evolution management. The investigation led to the realisation that there is a need to control how services are evolved without forcing customers to rebuild their consumer applications. One of the main technologies that consumers rely on is the WSDL file which is

known as the Technical Contract of a web service. Having control over how the Contract is designed ultimately means having control over how the web service is managed. It was established through a literature search that control over Contract design is achieved through DbC, hence this work also followed this approach in developing the service versions. Versioning of these contracts, however, can be done on the basis of established techniques of the organisation in question.

When an upgrade is made and a new web service version is implemented, the mechanism in place to ensure seamless transition of the web service and continuous service to older consumers relying on the old version is a contracts-aware proxy. This research identified that there was a lack of use of DbC and knowledge of the advantages DbC offers in managing the evolution of the web service. Consequently, this work coupled the DbC approach with contracts to develop a contract-based proxy model that transforms requests and responses according to the understanding between the service provider and the service consumer. The model ensures that old consumers remain functional for as long as needed while supporting the newer consumers with added functionality as required. Although, there is a small loss in web service throughput due to additional processing by the proxy implementation this research has succeeded in demonstrating that the evolution of web services can be managed without unnecessary disruptions to consumers and without forcing corporate or individual customers to rebuild their software or consumer applications, while at the same time reducing the amount of resources the service provider has to use and maintain.

Research sub-question 3: How can we identify the service version that is being requested by a consumer?

The SOA model brings with it a major challenge for service providers, and the nature of services in SOC does little to assist. Although there are no concrete web service versioning techniques, the contract can be used to identify the consumer. The 2nd research objective was achieved through

designing a proxy that could read the contract and compare the incoming requests before processing. The proxy receives the incoming SOAP request and searches for a Contract with matching attributes in order to determine the version implementation of the consumer requesting the service, since the Contracts are versioned. Only after the consumer's version identification process has been completed can the client be serviced with the compatible transformation to the running web service. The general practice used in versioning web services is creating an entirely new web service with its own namespace, which unfortunately entails the rebuilding of all consumers of that service. Other approaches were to version the contract, service and endpoint separately. However, the challenge lies in identifying who the set of users are of a particular web service offering. Looking at SOAP web services, the Contract stands as a binding agreement between a web service provider and the consumer willing to use the web service offering. Hence, understanding the contents of the Contracts enables the identification of consumers through matching the service requests to the requirements in the Contract.

Research sub-question 4: How can the proposed model be validated and what mechanisms or procedures can be used to evaluate the efficacy and utility of the proposed solution?

Seeing that this work was undertaken following the design science research methodology, it is only befitting that the model designed following design science be evaluated and validated following the same. The last research objective enabled us to develop the model and respond to the 4th research sub-question. Design science involves design for some artifact that aims at changing some phenomenon which is of some human necessity. Thus there is a need to evaluate the artifact to see if it does indeed address the problem.

The validation criteria of design science concern mainly the success of the artifact and its generality.

Success of the Artifact:

The success of the artifact can be measured in terms of its **usefulness**, where "useful" is the degree to which the artifact contributes to the achievement of a result. None of the results presented in this work were obtained outside the development of the design science research artifact. Thus the contracts-based proxy model was instrumental in achieving the workings of the instantiation of the artifact and consequently the results and conclusions which were drawn from the artifact. We can safely conclude that the artifact was very useful in contributing to the achievement of results presented in Chapter 6 of this work.

Success is also measured as the degree to which the artifact achieves the desired result, its **efficacy**. The model presented in Chapter 4 of this work was intended to manage the evolution process of a web service. This research demonstrated this management while applying it to the running scenario of the StockQuote web service described in Chapter 3. The theory around the designing of the web service is implementable and used in the industry to develop standards-based corporate web services today and this work successfully implemented an instantiation of the model (the artifact), and simulated the evolution of the web service from one version to the next while still supporting old consumers and reusing the same resources. The implementation of the model does not drastically affect the quality of service that is offered to consumers as all response times observed are well within acceptable industry standards. Therefore, the model successfully enables the evolution of SOAP Web Services with disrupting or degrading the service to consumers.

Generality:

Though this model has been applied in a particular scenario, that of the StockQuote web service, the model itself was developed to address evolution management challenges in web services.

Generalisation means that the applicability of the model is suited not only for the StockQuote web service used in this work's illustrations, but is expected to be universal. This work presented a model that is generally applicable to all SOAP based web services. In Chapter 2, this research identified that SOAP web services are classified under the E-type systems, which exhibit generality as one their characteristics.

7.3. Future work

One area that would need further exploration is in implementation of a buffer in the proxy to accommodate more traffic. During the experimentation, the errors that came in SoapUI were as a result of access-to-service denials. This was owing to the fact that the ESB was accepting a workload that was more than the individual services could handle synchronously. There is a need to eliminate the denial-of-service errors due to services being too busy to handle incoming requests. Queueing requests may be a possible solution; hence a possible extension of this work could be in researching which best queuing mechanisms can be implemented. Queuing may increase response times but that may be a more desirable outcome than a failed service request. Automation of proxy-contract acquisition from the local or remote registry is another area that needs to be explored. Instead of the current approach where configuration of the proxy is done manually in the event of a new version and availability of a new contract the proxy should be able to reconfigure itself and accommodate a new service version and automatically support both existing and new consumers. Another area of research lies in conducting an impact analysis study or developing a mechanism for performing impact analysis investigations to determine how a change will affect the service clients.

7.4. Contributions to knowledge

From the commencement of this work the goal was to come up with a contract-based model for managing the evolution of shared services. This work provides a solution that minimises disruptions to service consumers when a service changes by enabling continued support for older service versions in a single contract-based proxy while newer versions are implemented. Other solutions that have been put forward relatively increase the complexity of managing multiple

versioned services, and also need more resources to support the multiple versions running concurrently.

This work also presents a way of expressing the throughput loss as a percentage. This was given by Equation 6.1 in Chapter 6. This equation was derived from a similar notion around which income tax is charged on an individual's earnings. Contracts have been identified as a key component in web services, allowing for controlled development of web services. Other researchers have reasoned around the compatibility theories of web services using contracts. This work contributes the notion of actually using the contract in the process of managing the evolution of web services. This was demonstrated through the implementation of a proxy service that relied on these contracts to transform and serve consumers that requested for service from an older version that had otherwise been updated to a new version.

REFERENCES

Afshar, M., Cincinatus, M., Hynes, D., Clugage, K., Patwardhan, V., 2007. SOA governance: Framework and best practices. White Pap. Oracle Corp. May.

Airbnb [WWW Document], n.d. URL https://www.airbnb.com/about/about-us (accessed 10.7.15).

Akkiraju, R., Farrell, J., Miller, J., Nagarajan, M., Schmidt, M.-T., Sheth, A., Verma, K., 2005. Web Service Semantics-WSDL-S.

Alonso, G., Casati, F., Kuno, H., Machiraju, V., 2004. Web services. Springer.

Andrade, A., Luiz, J., 2000. Evolution by Contract, in: Proceeding of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications. Presented at the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota USA.

Andrikopoulos, V., Benbernou, S., Papazoglou, M.P., 2012. On the evolution of services. Softw. Eng. IEEE Trans. On 38, 609–628.

Bellahsène, Z., Léonard, M., 2008. Advanced Information Systems Engineering: 20th International Conference, CAiSE 2008 Montpellier, France, June 18-20, 2008, Proceedings. Springer.

Berners-Lee, T., 2009. Web Services overview - Design Issues [WWW Document]. URL http://www.w3.org/DesignIssues/WebServices.html (accessed 1.17.15).

Bernhardt, J., Seese, D., 2009. A Conceptual Framework for the Governance of Service-Oriented Architectures, in: Feuerlicht, G., Lamersdorf, W. (Eds.), Service-Oriented Computing – ICSOC 2008 Workshops, Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 327–338.

Bhuvaneswari, N.S., Sujatha, S., 2011. Integrating Soa and Web Services. River Publishers.

Bianco, P., Lewis, G.A., Merson, P., 2008. Service level agreements in service-oriented architecture environments [WWW Document]. URL http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA528751 (accessed 4.14.14).

Bloomberg, J., 2015. Service-Oriented Architecture: Enabler of the Digital World - Forbes [WWW Document]. URL http://www.forbes.com/sites/jasonbloomberg/2015/02/09/service-oriented-architecture-enabler-of-the-digital-world/ (accessed 9.27.15).

Boehm, B., W., 1981. Software engineering economics. Englewood Cliffs NJ Prentice-Hall 197.

Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., Orchard, D., 2004. Web Services Architecture [WWW Document]. Web Serv. Archit. URL http://www.w3.org/TR/ws-arch/ (accessed 12.11.15).

Bordeaux, L., Salaün, G., Berardi, D., Mecella, M., 2005. When are Two Web Services Compatible?, in: Shan, M.-C., Dayal, U., Hsu, M. (Eds.), Technologies for E-Services, Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 15–28.

Brown, W.A., Moore, G., Tegan, W., 2006. SOA Governance-IBM's approach. Somers NY.

Bryant, A., Kirkham, J.A., 1983. B. W. Boehm Software Engineering Economics: A Review Essay. SIGSOFT Softw Eng Notes 8, 44–60. doi:10.1145/1010891.1010897

Chiponga, K., Tarwireyi, P., Adigun, M.O., 2014a. A version-based transformation proxy for service evolution, in: 6th IEEE International Conference on Adaptive Science and Technology (ICAST) 2014. Presented at the ICAST 2014, Ota, Nigeria.

Chiponga, K., Tarwireyi, P., Adigun, M.O., 2014b. Contract-based Web Service Evolution Model. Presented at the SATNAC, Port Elizabeth, Eastern Cape.

Crocker, D., 2004. Safe object-oriented software: the verified design-by-contract paradigm, in: Practical Elements of Safety. Springer, pp. 19–41.

Daigneau, R., 2011. Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services, 1 edition. ed. Addison-Wesley.

Dlamini, S.W., Tarwireyi, P., Adigun, M.O., 2013. A Model-Driven Approach for Managing Variability in Service-Oriented Environments. Int. J. Inf. Technol. Amp Comput. Sci. IJITCS 012013 891-97 8, 91–97.

Erl, T., Karmarkar, A., Walmsley, P., Haas, H., Yalcinalp, L.U., Liu, K., Orchard, D., Tost, A., Pasley, J., more, & 6, 2008. Web Service Contract Design and Versioning for SOA, 1 edition. ed. Prentice Hall, Upper Saddle River, NJ.

Espinha, T., Chen, C., Zaidman, A., Gross, H.-G., 2012. Maintenance Research in SOA - Towards a Standard Case Study, in: 2012 16th European Conference on Software Maintenance and Reengineering (CSMR). Presented at the 2012 16th European Conference on Software Maintenance and Reengineering (CSMR), pp. 391–396. doi:10.1109/CSMR.2012.49

Espinha, T., Zaidman, A., Gross, H.-G., 2015. Web API growing pains: Loosely coupled yet strongly tied. J. Syst. Softw. 100, 27–43. doi:10.1016/j.jss.2014.10.014

Fokaefs, M., Mikhaiel, R., Tsantalis, N., Stroulia, E., Lau, A., 2011. An Empirical Study on Web Service Evolution, in: 2011 IEEE International Conference on Web Services (ICWS). Presented at the 2011 IEEE International Conference on Web Services (ICWS), pp. 49–56. doi:10.1109/ICWS.2011.114

Fokaefs, M., Stroulia, E., 2012. WSDarwin: automatic web service client adaptation, in: Jacobsen, H.-A., Zou, Y., Chen, J. (Eds.), Center for Advanced Studies on Collaborative Research, CASCON '12, Toronto, ON, Canada, November 5-7, 2012. IBM / ACM, pp. 176–191.

Frank, D., Lam, L., Fong, L., Fang, R., Khangaonkar, M., 2008. Using an Interface Proxy to Host Versioned Web Services, in: IEEE International Conference on Services Computing, 2008. SCC '08. Presented at the IEEE International Conference on Services Computing, 2008. SCC '08, pp. 325–332. doi:10.1109/SCC.2008.84

Göbel, H., Cronholm, S., 2012. Design science research in action - experiences from a process perspective.

González-Barahona, J.M., Robles, G., Herraiz Tabernero, I., Ortega, F., 2014. Studying the laws of software evolution in a long-lived FLOSS project. J. Softw. Evol. Process.

Gorinsek, J., Van Baelen, S., Berbers, Y., De Vlaminck, K., 2003. Managing quality of service during evolution using component contracts, in: Proc. 2nd International Workshop on Unanticipated Software Evolution, Warsaw, Poland. pp. 57–62.

Govardhan, S., Feuerlicht, J., 2009. SOA: Trends and Directions, in: Proceedings of the 17th International Conference on Systems Integration 2009. Presented at the 17th International Conference on Systems Integration, Prague, Czech Republic, pp. 149–154.

Guindon, C., n.d. Eclipse.org - Juno Simultaneous Release [WWW Document]. URL https://eclipse.org/juno/ (accessed 4.8.16).

Guo, S., Bai, F., Hu, X., 2011. Simulation software as a service and Service-Oriented simulation experiment, in: 2011 IEEE International Conference on Information Reuse and Integration (IRI). Presented at the 2011 IEEE International Conference on Information Reuse and Integration (IRI), pp. 113–116. doi:10.1109/IRI.2011.6009531

Gu, Q., Lago, P., 2011. Guiding the selection of service-oriented software engineering methodologies. Serv. Oriented Comput. Appl. 5, 203–223. doi:10.1007/s11761-011-0080-0

Hägg, S., Ygge, F., Gustavsson, R., Ottosson, H., 1996. DA-SoC: A testbed for modelling distribution automation applications using agent-oriented programming, in: Distributed Software Agents and Applications. Springer, pp. 63–76.

Haynes, S.R., Carroll, J.M., 2007. Theoretical Design Science in Human–Computer Interaction: A Practical Concern? Artifact 1, 159–171.

Hevner, A., Chatterjee, S., 2010. Design Research in Information Systems: Theory and Practice. Springer.

Hevner, A.R., March, S.T., Park, J., Ram, S., 2004. Design Science in Information Systems Research. MIS Q. 28, 75–105.

Hollunder, B., Herrmann, M., Hulzenbecher, A., 2012. Design by Contract for Web Services: Architecture, Guidelines, and Mappings. Iaria J. 5.

Jepsen, T., 2001. SOAP cleans up interoperability problems on the Web. IT Prof. 3, 52–55. doi:10.1109/6294.939937

Kajko-Mattsson, M., Lewis, G.A., Smith, D.B., 2007. A framework for roles for development, evolution and maintenance of soa-based systems, in: Proceedings of the International Workshop on Systems Development in SOA Environments. IEEE Computer Society, p. 7.

Kajko-Mattsson, M., Lewis, G.., Smith, D.B., 2008. Evolution and Maintenance of SOA-Based Systems at SAS, in: Hawaii International Conference on System Sciences, Proceedings of the 41st Annual. Presented at the Hawaii International Conference on System Sciences, Proceedings of the 41st Annual, pp. 119–119. doi:10.1109/HICSS.2008.154

Kaminski, P., Müller, H., Litoiu, M., 2006. A design for adaptive web service evolution, in: Proceedings of the 2006 International Workshop on Self-Adaptation and Self-Managing Systems. ACM, pp. 86–92.

Karus, S., 2007. Forward Compatible Design of Web Services Presentation Layer. Masters Thesis, Faculty of Mathematics & Computer Science, University of Tartu, Estonia, 2007. http://www. cyber. ee/dokumendid/Karus. pdf.

Kern, H., 2003. How to measure system availability targets [WWW Document]. TechRepublic. URL http://www.techrepublic.com/article/how-to-measure-system-availability-targets/ (accessed 9.1.15).

Khadka, R., Saeidi, A., Jansen, S., Hage, J., 2013. A structured legacy to SOA migration process and its evaluation in practice, in: Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), 2013 IEEE 7th International Symposium on the. IEEE, pp. 2–11.

Khan, M.W., Abbasi, E., 2015. Differentiating Parameters for Selecting Simple Object Access Protocol (SOAP) vs. Representational State Transfer (REST) Based Architecture. J. Adv. Comput. Netw. 3.

Kijas, S., Zalewski, A., 2013. Towards Evolution Methodology for Service-Oriented Systems, in: Zamojski, W., Mazurkiewicz, J., Sugier, J., Walkowiak, T., Kacprzyk, J. (Eds.), New Results in Dependability and Computer Systems, Advances in Intelligent Systems and Computing. Springer International Publishing, pp. 255–273.

Kontogiannis, K., Lewis, G.A., Smith, D.B., 2008. A Research Agenda for Service-oriented Architecture, in: Proceedings of the 2Nd International Workshop on Systems Development in SOA Environments, SDSOA '08. ACM, New York, NY, USA, pp. 1–6. doi:10.1145/1370916.1370917

Krai, J., Zemlicka, M., 2007. The Most Important Service-Oriented Antipatterns, in: International Conference on Software Engineering Advances, 2007. ICSEA 2007. Presented at the

International Conference on Software Engineering Advances, 2007. ICSEA 2007, pp. 29–29. doi:10.1109/ICSEA.2007.74

Lehman, M.M., 1980. Programs, life cycles, and laws of software evolution. Proc. IEEE 68, 1060–1076.

Levy, A., 2014. 200ms: The Magical Number for Faster Response Times [WWW Document]. Crittercism. URL http://www.crittercism.com/2014/03/200ms-the-magical-number-for-faster-response-times/ (accessed 11.24.15).

Lewis, G.A., Smith, D.B., 2013. Research Challenges in the Maintenance and Evolution of Service-Oriented Systems. Migrating Leg. Appl. Chall. Serv. Oriented Archit. Cloud Comput. Environ. 13–39. doi:10.4018/978-1-4666-2488-7.ch002

Lewis, G., Morris, E., Smith, D., 2005. Service-oriented migration and reuse technique (smart), in: Software Technology and Engineering Practice, 2005. 13th IEEE International Workshop on. IEEE, pp. 222–229.

Lewis, G., Smith, D., Kontogiannis, K., 2010. A Research Agenda for Service-Oriented Architecture (SOA): Maintenance and Evolution of Service-Oriented Systems. Softw. Eng. Inst.

Lippert, S.K., Govindarajulu, C., 2015. Technological, organizational, and environmental antecedents to web services adoption. Commun. IIMA 6, 14.

Load Testing Overview | Load Testing [WWW Document], n.d. URL http://www.soapui.org/load-testing/concept.html (accessed 7.13.15).

Malik, Z., Medjahed, B., 2010. Trust Assessment for Web Services under Uncertainty, in: Maglio, P.P., Weske, M., Yang, J., Fantinato, M. (Eds.), Service-Oriented Computing, Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 471–485.

Mallayya, D., Ramachandran, B., Viswanathan, S., 2015. An Automatic Web Service Composition Framework Using QoS-Based Web Service Ranking Algorithm. Sci. World J. 2015.

Mingyan, Z., Yanzhang, W., Xiaodong, C., Kai, X., 2008. Service-Oriented Dynamic Evolution Model, in: International Symposium on Computational Intelligence and Design, 2008. ISCID '08. Presented at the International Symposium on Computational Intelligence and Design, 2008. ISCID '08, pp. 322–326. doi:10.1109/ISCID.2008.147

Moha, N., Palma, F., Nayrolles, M., Conseil, B.J., Guéhéneuc, Y.-G., Baudry, B., Jézéquel, J.-M., 2012. Specification and detection of SOA antipatterns, in: Service-Oriented Computing. Springer, pp. 1–16.

Mosser, S., Blay-Fornarino, M., 2013. "Adore", a logical meta-model supporting business process evolution. Sci. Comput. Program. 78, 1035–1054.

Mulligan, G., Gracanin, D., 2009. A comparison of SOAP and REST implementations of a service based interaction independence middleware framework, in: Simulation Conference (WSC), Proceedings of the 2009 Winter. Presented at the Simulation Conference (WSC), Proceedings of the 2009 Winter, pp. 1423–1432. doi:10.1109/WSC.2009.5429290

Murugesupillai, E., Mohabbati, B., Gašević, D., 2011. A Preliminary Mapping Study of Approaches Bridging Software Product Lines and Service-oriented Architectures, in: Proceedings of the 15th International Software Product Line Conference, Volume 2, SPLC '11. ACM, New York, NY, USA, pp. 11:1–11:8. doi:10.1145/2019136.2019149

Olivier, M.S., 2009. Information technology research: a practical guide for computer science and informatics, 3rd ed. Van Schaik.

Papazoglou, M.P., 2008a. Web Services: Principles and Technology. Pearson Prentice Hall.

Papazoglou, M.P., 2008b. The Challenges of Service Evolution, in: Bellahsène, Z., Léonard, M. (Eds.), Advanced Information Systems Engineering, Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 1–15.

Perez, S., 2015. Facebook Wins "Worst API" in Developer Survey [WWW Document]. TechCrunch. URL http://social.techcrunch.com/2011/08/11/facebook-wins-worst-api-in-developer-survey/ (accessed 10.19.15).

Ren, M., Lyytinen, K.J., 2008. Building enterprise architecture agility and sustenance with SOA. Commun. Assoc. Inf. Syst. 22, 4.

Robak, S., Franczyk, B., 2003. Modeling Web Services Variability with Feature Diagrams, in: Chaudhri, A.B., Jeckle, M., Rahm, E., Unland, R. (Eds.), Web, Web-Services, and Database Systems, Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 120–128.

Roetzheim, W.H., 2000. Estimating Software Costs. Softw. Dev.-San Franc.- 810 8, 66–68.

Ruz, C., Baude, F., 2010. Enabling SLA monitoring for component-based SOA applications, in: 36th Euromicro Conf. on Software Engineering and Advanced Applications. Presented at the 36th Euromicro Conf. on Software Engineering and Advanced Applications, Lille, France.

Schepers, T.G.J., Iacob, M.E., Van Eck, P.A.T., 2008. A Lifecycle Approach to SOA Governance, in: Proceedings of the 2008 ACM Symposium on Applied Computing, SAC '08. ACM, New York, NY, USA, pp. 1055–1061. doi:10.1145/1363686.1363932

Sinnema, M., Deelstra, S., Nijhuis, J., Bosch, J., 2004. Covamof: A framework for modeling variability in software product families, in: Software Product Lines. Springer, pp. 197–213.

Sommerville, I., 1982. Software Engineering, 8th ed.

Sun, C., Rossing, R., Sinnema, M., Bulanov, P., Aiello, M., 2010. Modeling and managing the variability of Web service-based systems. J. Syst. Softw. 83, 502–516. doi:10.1016/j.jss.2009.10.011

Tiago Espinha, C.C., 2012. Spicy stonehenge: Proposing a SOA case study 57–58. doi:10.1109/PESOS.2012.6225940

Treiber, M., Truong, H.-L., Dustdar, S., 2008. SEMF - Service Evolution Management Framework, in: Software Engineering and Advanced Applications, 2008. SEAA '08. 34th Euromicro Conference. Presented at the Software Engineering and Advanced Applications, 2008. SEAA '08. 34th Euromicro Conference, pp. 329–336. doi:10.1109/SEAA.2008.44

Tsai, W.T., Paul, R., Song, W., Cao, Z., 2002. Coyote: an XML-based framework for Web services testing, in: 7th IEEE International Symposium on High Assurance Systems Engineering, 2002. Proceedings. Presented at the 7th IEEE International Symposium on High Assurance Systems Engineering, 2002. Proceedings, pp. 173–174. doi:10.1109/HASE.2002.1173120

Tsalgatidou, A., Pilioura, T., 2002. An Overview of Standards and Related Technology in Web Services. Distrib. Parallel Databases 12, 135–162. doi:10.1023/A:1016599017660

Vara, J.M., Andrikopoulos, V., Papazoglou, M.P., Marcos, E., 2012. Towards Model-Driven Engineering Support for Service Evolution. J UCS 18, 2364–2382.

Wagh, K., Thool, R., 2012. A Comparative Study of SOAP Vs REST Web Services Provisioning Techniques for Mobile Host. J. Inf. Eng. Appl. 2, 12–16.

Wala, T., Sharma, A., K., 2014. A Comparative Study of Web Service Testing Tools 4, 257–261.

Web Services Architecture [WWW Document], n.d. URL http://www.w3.org/TR/2002/WD-ws-arch-20021114/ (accessed 8.7.15).

Weyuker, E.J., Vokolos, F.I., 2000. Experience with performance testing of software systems: issues, an approach, and case study. IEEE Trans. Softw. Eng. 26, 1147–1156. doi:10.1109/32.888628

What is soapUI? | About SoapUI [WWW Document], n.d. URL http://www.soapui.org/about-soapui/what-is-soapui-.html (accessed 3.9.15).

Wilde, E., 2004. Semantically extensible schemas for web service evolution, in: Web Services. Springer, pp. 30–45.

Wilde, N., Coffey, J., Reichherzer, T., White, L., 2012. Open SOALab: Case study artifacts for SOA research and education. IEEE, pp. 59–60. doi:10.1109/PESOS.2012.6225941

Witte, A.G., 2013. SOA Governance in the Low Countries.

Zuo, W., Benharkat, A.N., Amghar, Y., 2014a. Change-centric Model for Web Service Evolution, in: 2014 IEEE International Conference on Web Services (ICWS). Presented at the 2014 IEEE International Conference on Web Services (ICWS), pp. 712–713. doi:10.1109/ICWS.2014.111

Zuo, W., Nabila, A., Benharkat, Y., 2014b. Holistic and Change-centric Model for Web Service Evolution, in: Fourth International Workshop on the Future of Software Engineering For/in the Cloud. Presented at the Dans 2014, Alaska. doi:10.1109/SERVICES.2014.51