HEALTH INFORMATION SYSTEMS INTEROPERABILITY IN AFRICA

Service Oriented Architectural Model for interoperability in African context

by

BONIFACE KABASO

Thesis submitted in fulfilment of the requirements for the degree

Doctor of Technology: Information Technology

in the Faculty of Informatics And Design

at the Cape Peninsula University of Technology

Supervisor: Professor Mikko Korpela (2010-2014)

Supervisor: Professor Vesper Owei (2008-2010)

Cape Town

August 2014

DECLARATION

I, Boniface Kabaso, declare that the contents of this dissertation/thesis represent my own unaided work, and that the dissertation/thesis has not previously been submitted for academic examination towards any qualification. Furthermore, it represents my own opinions and not necessarily those of the Cape Peninsula University of Technology.

_____

Signed **Date**

Abstract

Africa has been seeing a steady increase in the Information and Communication Technology (ICT) systems deployed in health care institutions. This is evidenced by the funding that has been going into health information systems from both the government and the donor organisations.

Large numbers of national and international agencies, research organisations, Non-Governmental Organisations(NGOs) etc continue to carry out studies and develop systems and procedures to exploit the power of Information and Communication Technology (ICT) in public and private health institutions.

This uncoordinated mass migration to electronic medical record systems in Africa has created a heterogeneous and complex computing environment in health care institutions, where most of the deployed systems have technologies that are local, proprietary and insular.

Furthermore, the electronic infrastructure in Africa meant to facilitate the electronic exchange of information has a number of constraints. The infrastructure connectivity on which ICT applications run, is still segmented. Most parts of Africa lack the availability of a reliable connectivity infrastructure. In some cases, there is no connectivity at all.

This work aims at using Service Oriented Architectures (SOA) to address the problems of interoperability of systems deployed in Africa and suggest design architectures that are able to deal with the state of poor connectivity.

SOA offers to bring better interoperability of systems deployed and re-usability of existing IT assets, including those using different electronic health standards in a resource constrained environment like Africa.

Acknowledgements

Work like this never gets produced without a huge input from a lot of people who worked tirelessly to make it happen. Here is a list of these people:

First of all, special thanks to Professor Mikko Korpela for not only stepping up to supervise this work, but also being there to put on me the right amount of pressure to get me working to finish this.

Also special thanks to the late Professor Vesper Owei who kept hammering in the importance of rigour in research. I never got it at the time. I now get it and thanks for planting that in my mind at an early stage.

An enormous thanks to Professor Retha De La Harpe for working hard to create those opportunities where my workload was reasonable so I could work. Also thanks for the Incubation Hub where a significant portion of this work was tested and applied.

Professor Bennett Alexander, thanks for the support, especially during the write up of this work when it was realised that I was not entitled to any study leave or sabbatical to finish this work.

I would be remiss if I did not mention the input of B-Tech and Diploma students who worked hard to code some APIs and put them to test to see if they worked or not. This space is not enough to list all your names. I believe working on projects that fed into this work gave you the technical knowledge and experience that you would value for the rest of your programming life.

No proprietary software was used in the production of this thesis. All the tools used in putting this thesis together are free and open source, with the main software being LaTex

Contents

List of Figures

List of Tables

Abbreviations

API	Application Programming Interface
ART	Anti Retroviral Treatment
ARTMS	ART Medical System
ARV	Antiretroviral Drugs
CDA	Clinical Document Architecture
CDA App	Clinical Data Application
CDT	Current Dental Terminology
CORBA	Common Object Request Broker
CPT	Current Procedural Terminology
DCOM	Distributed Component Object Model
DICOM	Digital Imaging and Communications in Medicine
D-MIN	Domain Message Information Model
DSM	Diagnostic and Statistical Manual of Mental Disorders
EAI	Enterprise Application Integration
ECRTC	Eastern Cape Regional Training Center
ESB	Enterprise Service Bus
FTP	File Transfer Protocol
HATEOAS	Hypermedia As The Engine Of Application State
HCPCS	health care Common Procedure Coding System
HI	Health Informatics
HL7	Health Level Seven International
HMD	Hierarchical Message Description
HTML	Hyper Text Markup Language
HTTP	Hyper Text Transfer Protocol
ICD	International Classification of Diseases

ICT	Information and Communication Technology
IHE	Integrating the Healthcare Enterprise
ISO	Internation Standard Organisation
JMS	Java Messaging Service
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LOINC	Logical Observation Identifiers, Names, and Codes
MML	Medical Markup Language
MOM	Message Oriented Middleware
NDC	National Drug Codes
RDR	Runtime Dynamic Routing
REST	Representational State Transfer
RFC	Request For Comments
RID	Retrieve Information for Display
RIM	Reference Information Model
RMI	Reomote Method Invocation
R-MIN	Refined Message Information Model
SOAP	Simple Access Object Protocol
SMTP	Simple Mail Transfer Protocol
SOA	Service Oriented Architecture
SOA-DDA	SOA Driven Disconnected Architecture
SNOMED CT	Systematised Nomenclature of Medicine Clinical Terms
SR	Structured Reporting
URI	Universal Resource Identifier
WG	Working Group
WSDL	Web Services Description Language
XDS	Cross-Enterprise Document Sharing
XML	Extensible Markup Language

Chapter 1

Introduction

Most health care institutions worldwide are in the core business of providing health care to patients. Patients, over the course of their life, visit several health institutions and their medical records are captured and stored either on paper based systems or digital electronic systems. For an institution to provide good continuity of care to its patients, it needs a fast, easy and 360 degrees view into a patients complete medical history. However, the records that make up this history are scattered around several institutions that the patient has visited in his or her life time (Mandl et al., 2001). Getting to this information is a great challenge, particularly if most of the information is in paper based systems.

The good news is that most institutions have realised the need to digitise medical records and efforts are under way in most countries funded by vendors, donors and national governments (Fraser et al., 2005).

Despite this mass migration to digital systems, the problem of accessing patient records across departments, institutions, regional or international boundaries still remains a problem to be solved. Health institutions' computing environment is very heterogeneous and complex (Grimson et al., 2000). Most electronic records are kept in various types of digital systems ranging from flat files to databases based on home grown or off the self systems.

Most of these systems have technologies that are local, proprietary and insular. They are not designed to communicate with other systems whether inside or outside individual health institutions. Their primary design is to capture and store patient records for local stakeholders.

There is now a realisation of the importance of sharing clinical information(Walker et al., 2005). Information coordination within an institution and across various health institutions is essential in ensuring that care providers work together effectively to support integrated care initiatives that deliver the best levels of service and efficiency (Dogac et al., 2006).

At this stage, in order to streamline and coordinate access to this information and eliminate the interoperability barriers, it is not practical to either discard the existing systems or migrate to the new systems. The deployed systems have an enormous amount of data stored across disparate pieces of software. Most of the health institutions have made extensive investments in these system resources over the course of many years. The solution to these issues must, therefore, capitalise on the investments made in the existing old systems.

The big question asked worldwide is how this can be done in a cost-effective way. It is more practical and cost-effective to evolve and enhance these deployed systems than rip and replace them. Service Oriented Architecture (SOA) provides a cost-effective solution compared to the rip and replace strategy.

SOA is emerging as the premier integration and architecture framework in today's complex and heterogeneous computing environment. SOA applications stretch across infrastructure tiers, software, middleware, operating systems, institution, regional or national borders (Lublinsky and Tyomkin, 2003).

1.1 Research Fields

The research in this study is anchored on three fields listed below. It is important to clarify some of the terms and concepts that are relevant to this work from these fields of research at this stage.

- Information Systems.
- Health Informatics.
- Software Engineering.

1.1.1 Information Systems

There is a myriad definitions of what constitutes an Information System. For the purpose of this work, the definition used by Gupta (Gupta, 1989) that an Information System consist of computer software systems that provide information to aid an organisation's decision making will be maintained. In other words, an information system creates, processes, stores and generates information to help individuals or organisations make meaningful decisions.

As further pointed out by several other information systems practitioners, an information system in an organisation provides facts which are to be exploited by the human users of the system in order to function effectively. (Some of these view points will be reviewed in chapter 2 on literature review).

Information systems are the software and hardware systems that support data-intensive applications.

1.1.2 Health Informatics

According to the Stanford Medical Informatics, Health or Medical Informatics is the Scientific field that deals with biomedical information, data, and knowledge - their storage, retrieval, and optimal use for problem solving and decision making. This definition accordingly touches on all basic and applied fields in biomedical science and is closely tied to modern information technologies, notably in the areas of computing and communication.

Health Informatics is the logic of Health care. It is the rational study of the way individuals or organisations technologically look at patients, and the way that treatments are defined, selected and evolved. It is the study of how clinical knowledge is created,

shaped, shared and applied. Ultimately, it is the study of how we organise systems to create and run health care organisations.

Although the name Health Informatics only came into use around 1973 (Hovenga, 2004), it is a study that is as old as Health care itself.

Health Informatics has grown considerably as a clinical discipline in recent years fuelled by the advances in computer technology. What has fundamentally changed is our ability to describe and manipulate health knowledge at a highly abstract level, as has our ability to build up rich communication systems to support the process of health care.

1.1.3 Software Engineering

The Software Engineering definition in this work will be based on (Ghezzi et al., 2002) where emphasis is on design, specification, verification, production, management and tools involved in the production of sound and quality software artefacts. More focus is placed on Distributed Systems.

A Distributed System is a system consisting of a collection of autonomous machines connected by communication networks and equipped with software systems designed to produce an integrated and consistent computing environment.

The key purposes of the distributed systems can be represented by: Resource Sharing, Openness, Processing and Concurrency, Scalability, Fault-Tolerance and Transparency (Balter et al., 1991).

Resource Sharing in a Distributed System allows the resources - hardware, software and data to be easily shared among users.

Openness of Distributed Systems is achieved by specifying the key software interface of the system and making it available to software developers so that the system can be extended in many ways.

The Processing and Concurrency can be achieved by sending requests to multiple machines connected by networks at the same time.

A Scalable Distributed System running on a collection of a small number of machines can be easily extended to a large number of machines to increase the processing power.

Machines connected by networks can be seen as redundant resources. A software system can be installed on multiple machines so that in the face of hardware faults or software failures, the faults or failures can be detected and tolerated by other machines.

Distributed systems can provide many forms of transparency such as Location Transparency (allowing local and remote information to be accessed in a unified way), Failure Transparency (enabling the masking of failures automatically) and Replication Transparency (allowing the duplicating software/data on multiple machines invisibly).

This work will mostly focus on the resource sharing, openness and scalability aspect of distributed systems.

1.2 Research Domain

The research domain of this work will be health care. The work will look at the health information systems that are deployed in the health care industry in Africa. Systems used will be mainly those that are being used for patient management and reporting. Systems that are being phased out for a better system will not be considered as this work is aimed at looking at functional systems that operate by sharing information in a Distributed Environment.

1.3 Research Motivation: Health Information Systems need Interoperability

The realities of Interoperability and re-usability problems have started to become more prominent in Africa as more systems are developed and deployed. The idea of having one system in place to solve this problem is not practical. The Service Oriented Architecture (SOA) approach, based on web services, gives a lot of promise. There has been so much hype and optimism about SOA and this is not totally misplaced. The

ideas of SOA are not new (Tsai, 2005). The key difference today that makes SOA generate the kind of coverage it is receiving is the current infrastructure which is in place (Natis, 2003).

We now have the infrastructure that gives us new ways to create, manage, share,and secure applications. By creating connections among disparate applications, not previously possible due to their inherent incompatibility, the SOA approach can help institutions increase the flexibility of their clinical processes, strengthen their underlying Information Technology (IT) infrastructure, and retain and re-use the IT services they have already deployed. These connections can allow a complete clinical system to be linked to the exact IT components needed to execute the processes. Furthermore, these connections can be mixed, matched and re-used to address problems unique to individual institutions. Since the services built around web services are based on industry standards (Papazoglou and Georgakopoulos, 2003), they can be shared with other institutions.

Additionally, within an institution, using SOA to connect information from disparate medical systems can give caregivers secure access to multiple applications, including lab reports, medical imaging, scheduling and other critical information about a patient. Since it integrates and synchronises patient information from disparate applications, clinical efficiency can be increased dramatically (Chen and Tsai, 2008).

Systems that are in use today do not need to be ripped up and thrown away. What they need, in addition to other standards issues (Eichelberg et al., 2005), is an upgrade in order to expose them as web services and make them part of a SOA. The challenge is to define what those services are, what standards to use and how to resolve various technical issues of interoperability (Dogac et al., 2006).

1.4 Research Problems

From the discussion in the introductory paragraph, it can be seen that most of the systems installed in the health care institutions do not have the means to connect to each other and later on exchange any information in a usable way due to a number of reasons.

Most of these systems run on different hardware, operating systems and network infrastructure. The programming languages used are different. The coding standards and message exchange protocols are different. All this makes the achievement of interoperability among these disparate systems (systems created independent of each other) a challenging task worth researching.

The core problem at the heart of this work is Interoperability of various IT assets in and outside a health institution and the connectivity of IT systems in the face of unreliable connectivity infrastructure in Africa.

1.4.1 Interoperability

According to the National Alliance for Health Information Technology (NAHIT, USA)(Detmer, 2003) interoperability in health care is defined as the ability of different information technology systems and software applications to communicate, exchange data accurately, effectively and consistently, and to use the information that has been exchanged.

An effective health information systems infrastructure requires cooperation between systems. Each system in the infrastructure requires information from one or more systems.

The deployed applications in health institutions present challenges on two fronts, namely the coding of stored data and the exchanges of this stored information.

Information management in the health care industry is very complex and very susceptible to errors because of the nature of the use cases. Most use cases involve the capturing of information that is required to be used by other stakeholders.

For example, the process of admitting a patient creates a lot of opportunities for errors to occur. Several stakeholders differ in the way they identify the players involved in the process. Even the coding of diseases and lab information is different. Several ways in which applications capture this data makes it incompatible with others. These ways vary from the use of international standards to use of proprietary free text to code the information.

There are various standards in which captured information is described and identified. These include, among many others, International Classification of Diseases (ICD), Current Procedural Terminology (CPT), Health Care Common Procedure Coding System (HCPCS), National Drug Codes (NDC), Current Dental Terminology (CDT), Systematised Nomenclature of Medicine Clinical Terms (SNOMED CT), Diagnostic and Statistical Manual of Mental Disorders (DSM), Logical Observation Identifiers, Names, and Codes (LOINC).

In Africa, most of the applications capture data using free text. This puts the chances of compatibility on a very low side. There is a high probability of fragmented systems as they are developed independently and without any co-ordination with the players in the industry.

The exchange of messages by applications presents even bigger challenges. To be able to exchange information among heterogeneous health care information systems, messaging interfaces need to be used. Typically, a messaging interface gathers data from one application, encodes the data into a message, and transmits the data over a network to another application. On the receiver side, the received messages are decoded, processed and the data which has been received is fed into the application to be stored and processed.

When proprietary formats are used in messaging, the number of the interfaces to be developed increases drastically. It would require the building of interfaces for every application that is deployed to archive Interoperability (Mattsson et al., 1999).

There are two aspects involved in the exchange of information-syntactic and semantic (Mead et al., 2006). Syntactic exchange involves the ability of two or more systems to see each other across the network and exchange messages. Syntactic exchange guarantees the message to be delivered but does not guarantee that the content of the message will be processed at the receiving end. To guarantee message content exchange, either the message content should conform to the receiving applications standard or semantic exchange must be provided.

Semantic exchange is the ability for information shared by systems to be understood and used by the receiving application. Most applications are designed with different semantics. For example, if we take a look at applications that collects information about

the age of patients, different systems will ask this question and record the answer differently as shown in table 1.1. The metric being measured is the same.

	Variable or data field for question	Variable or data field answer
System 1	Date of Birth	13th November 1971
System 2	How old is Patient	41 years
System 3	Patient age	41
System 4	Year of Birth	1971

TABLE 1.1: Table showing same information about the age of a patient from 4 systems

The four systems listed above record the same information differently. Making these systems exchange and use the exchanged information in a meaningful way will certainly pose some challenge.

1.4.2 Connectivity Infrastructure in Africa

The other core problem deals with the constrained African IT environment in terms of reliable connectivity. A truly functional networked health information system application requires a robust network infrastructure if it is to function well. Africa's network infrastructure is fraught with reliability problems caused by a number of factors. Most parts of Africa lack the availability of a reliable connectivity infrastructure (Afullo, 2000). Collection and consolidation of data from various places will likely remain a challenge for a long time.

It is, therefore, imperative that the solutions providers of IT systems take this into consideration and design application architectures that can function well in an environment where there is an unreliable network connection between the systems or nodes in a network topology.

Africa has a number of challenges that affect the connectivity infrastructure such as lack of electricity, guaranteed quality of service and robust regulatory framework. Any solution needs to address this African context.

1.5 Research Questions

The fundamental research questions that this work aims to address are:

1. What new piece of software is needed to build a bridge to mediate conversation between the disparate information systems that have been deployed in health institutions?
2. What new application framework can we build that will be used to construct new information systems applications that adhere to interoperability standards?
3. What solution can we propose that is well suited to the African or developing country landscape where the communication infrastructure is poor?
4. Can the same aspect of the solution of the design be demonstrated on two or more applications that have a wide deployment and fully exhibit the characteristics of the problems described in this proposal?
5. What lessons can we learn in the process of building this innovation and can we apply the lessons to similar future cases across Africa?

1.6 Goal

The goal of the questions is to set the stage that can help us come up with solutions derived from a clear picture of what is on the ground and build the innovation that will address the problems of interoperability of health information systems in Africa.

1.7 Purpose

The purpose of this research is to produce an outcome that can be generalised and applied to other cases in Africa. This work will have two outcomes. The first contribution of this study will be a set of open application interfaces or software that will make it possible to achieve integration and interoperability. The second will be the methodology or process that will go into tackling and solving the kind of problems that will be identified by the literature reviewed.

1.8 Scope

This study will be largely driven by the bottom up approach. The research will be scoped around open source software applications that are deployed and exhibit the characteristics described in the problems section. Commercial applications will be excluded from this work because it is extremely difficult to get valuable information from the private or commercial vendors. Private vendors give away little information about their systems in order to protect their niche markets and proprietary technologies which play a pivotal role in their endeavour to get a commercial edge over their fellow competitors. On the other hand, open source systems are freely available for study and analysis.

1.9 Research Methodology

While the problem definition of this work has its origin rooted in the social science world, the solution required for this problem is such that constructive research is required to produce the solution. The research approach will be based on the model proposed by (March and Smith, 1995). A two dimension framework is proposed in this paper where the authors state that research in IT must address design tasks faced by practitioners. The first dimension is based on Constructive Science Research (CSR) outputs of constructs, models, method and instatiations. The second is based on the research activities of Natural Science (NS) namely build–evaluate and theorise–justify.

The problem described in this work require the building of an artefact to solve the problem identified using specific software engineering methods. This type of work falls under the science of the artificial described in a book by (Simon, 1996). The products or the world created will further be studied using natural science research activities of theorise and justify. Social science research can further be done on the product produced in terms of how it performs in the environment it is deployed, but this will be beyond the scope of this work and reserved for another study.

1.10 Research Evaluation Criteria

This work will be required to produce constructs, models, methods and instatiations in the process of building the solution. The first issue to address will be whether the artefact produced work in the environment they are deployed. This will be a cyclical process that will eventually yield the desired artefact. The second issue will be to determine if the artefact produced has made the users and participants in the environment make progress to the point where the existing products can be replaced with the better ones as described by (March and Smith, 1995).

Metrics of what is trying to be accomplished will be defined by the questions asked and the artefact evaluated to see if the questions asked have been answered when the artefact is deployed in the African environment.

Further evaluation will be made to see if the built artefact was built according to the known standard knowledge and methods of constructing scalable software artefacts

Please note that evaluation of this work will be largely based in the technology sphere rather than the social sphere which is beyond the scope of this work.

1.11 Case Studies

We need to test or evaluate the utility of the products produced by this work. For this purpose, we apply our proposed solutions in three case studies chosen from different hospital environment domains and based on open source.

The criteria for the case studies will be based on the existence of two or more systems that are useful and need to interoperate. Cases where one system is being phased out for another will not be considered.

Enhancing an existing system and building a new system from scratch will also be used to demonstrate the practicality of the proposed solutions.

d

Chapter 2

Literature Review

2.1 Introduction

In this chapter, the relevant literature surrounding the problem discussions which forms the basis for the foundation of this study will be described. We shall briefly discuss and review some aspects of knowledge from some of the work that has been done. We are also going to look at the work that has been ongoing in the various fields that directly affects our study.

The sources of knowledge will be classified under the following subjects:

- Interoperability in Distributed Systems.
- Health Information Standards.
- ISO TC 215 on Health Informatics.
- Information Systems Infrastructure in Africa.

Interoperability in Distributed Systems will be used to explore the history of the challenges faced in integrating IT systems. Health information standards will categorise and detail the standards used in health care information systems in the world. Information about what is being done to deal with a myriad of health standards in information systems will be covered in the section on ISO TC 215 role in health informatics. Lastly, on the topic health information systems in Africa, we shall look at the landscape of information systems infrastructure in Africa.

2.2 Interoperability in Distributed Systems

2.2.1 Background

Interoperability problems are not unique to the health care sector. These problems have plagued the IT industry since the industry started building networked applications (Leiner et al., 1997). For decades, IT departments have laboured toward the goal of achieving efficiency for the benefit of their organisations, offering the flexibility to adapt to changing business needs rather than constraining them. For the most part, these aspirations have gone largely unrealised because of the result of incompatible systems and architectures that harden business processes rather than free them to become agile and responsive to the changing business needs.

Looking at the background of networked applications (Birman, 1997), it can be noted that within most institutions, specialised software applications are introduced through either homegrown efforts or third-party packaged software. These applications rapidly become interdependent, needing one another to complete basic processes such as admitting a patient. Consequently, the optimisation of processes soon demand interaction of not only systems within an institution, but also systems of other institutions and government agencies.

While interoperability between these systems has long been desired, the costs and complications to achieve this has resulted in most organisations containing hundreds of silos of information, with each silo being a separate and insular collection of data. Often these silos can only be connected at an enormous cost and effort (Goodhue et al., 1992).

2.2.2 What is Distributed Computing?

Distributed Computing is a term that describes hardware and software architecture where more than one computer participates in the completion of a given task (Garg, 2002).

For example, if a hospital has a new web based system used for processing patients appointments and an older mainframe computer that holds the complete details of patients and doctors, the two systems will need to collaborate in order to accomplish a common task such as appointment processing. To process an appointment, the web based system needs to look up and collect the details of patients and doctors held in the old mainframe system. The information needed to schedule the appointment is said to be distributed across the two systems. To complete the appointment, the old mainframe system must help the new web based system by sharing information from its database.

Certainly, it is possible to just load all the information in the old system onto the new system by using a process called consolidation and eliminate the need for this distributed architecture. However, the migration of data is time consuming, error-prone and expensive. The business logic in old systems is also byzantine(excessively complicated) and elaborate, often having grown over decades and gone through hundreds of human hands.

Furthermore, it is likely that under certain circumstances, old systems know how to process certain information based on particular codes, business practice or rule. Often this logic has been created in outdated languages that demand rare skills or the actual people who wrote the code to interpret the undocumented functions. To understand and reverse engineer the legacy code, which is often working just fine, is likely to be a difficult task.

In many situations, it is preferable, from a budgeting and operating perspective, to leave existing systems alone and arrange for them to operate in a distributed fashion (Thiran et al., 2006) with the new system. Additionally, the distributed transaction may involve systems outside an institution control. This could render any consolidation out of the question.

2.2.3 The Problems of Interoperability

The interoperability of two IT systems can generate the need for a lot of work in order for them to function. And when anything changes, the whole collaboration breaks down again, demanding new work to realign the participants.

In general, computers that come from different manufacturers running different operating systems with different software written differently by different teams do create a barrier for interoperability. They essentially do not understand and transmit information in ways that are compatible with other computers powered by different software. To communicate, the computers require the services of a mediator that speaks both languages.

The mediator interprets and re-transmits messages travelling between two computers operating in a distributed manner. The mediator must process each request for information twice. Once on its way from the requesting computer to the source of information and then again on its way back. If the interface were not available, the computers would not be able to communicate.

Proprietary protocols

Traditionally, most interfaces have relied on proprietary messaging protocols and data management standards (West, 2003). Interfaces tend to be custom coded, unique to the project at hand and unsuitable for reuse by other systems in the future. These interfaces tend to have limited challenges when just used to connect two computers. Connecting more than one computer in order to interoperate in a distributed environment reveals problems of having proprietary interfaces.

Tight coupling

The proprietary connection between systems creates a condition known as tight coupling (Dhama, 1995). The systems are bound to each other through the structures of the custom made mediation layer. Although there is nothing intrinsically wrong with tight coupling of systems in a distributed environment, when one wants to make a change, it is time consuming and costly to connect additional systems to the mediation layer or to change the systems that are currently connected. Each additional system will require its own interface to transmit and receive messages. As the number of systems in the configuration grows, so does the complexity of the mediation layer. Each system type has its own unique mode of communicating with the interface.

Change

As stated by (Beck and Andres, 2004) in their book, change is always guaranteed and it can be an expensive proposition in a tightly coupled environment. If the mediation layer

is based on custom developed systems, a software developer will have to implement any changes by rewriting the interface software.

The more extensive the change, the more complex the changes to the interface will likely be. The cost of maintaining and changing the interface can be substantial. The problem becomes even more complex if the interface is written in a programming language which has a small pool of skilled people, particularly in Africa. The costs of maintaining it may be even high and the interface may dangerously become dependent on certain few individuals.

These problems are simply the result of a condition where institutions needed to use numerous disparate systems to get tasks done and no common standards exists to make the process efficient.

2.2.4 Web Services Overview

Web services have become a standard way to achieve interoperability between systems as stated by (Yu et al., 2008). There are many different definitions available for a web service. In this study we shall go with the definition used by (Brown et al., 2002) in their paper. The definition is from World Wide Web Consortium (W3C) and defines a web service as follows:

A Web service is a software system identified by a Uniform Resource Identifier (URI) whose public interfaces and bindings are defined and described using XML (specifically WSDL). Its definition can be discovered by other software systems. These systems may then interact with the web service in a manner prescribed by its definition, using XML-based messages conveyed by Internet protocols

Simply put, a web service is a software component that provides a business function as a service over the internet that can be accessed through an address. Web services are next generation web applications, modules or components that can be thought of as a service provided over the internet (Yu et al., 2008). A web service component provides only business service, usually in the form of raw XML data that can be digested by virtually all client systems. A web service can be thought of as a self contained, self describing and modular application that can be published, located and invoked across the web.

Characteristics of Web Services

Web services behave differently from traditional software. They are based on open standards (Curbera et al., 2002) whereas most software in old paradigm connects through proprietary technologies. In addition, web services are loosely coupled when compared to traditional distributed computer systems. Once a piece of software has been exposed as a web service, it is relatively simple to move it to another computer because such services abstract the software functionality from the interface. Once a software program is available as a web service it can be accessed through open standards used in health information systems by the internet protocols.

Furthermore, like the internet, web services offer total network transparency in their implementation and use as described by (Sheltzer and Popek, 1986) in their study. Web services offer total network transparency because the coupling among web services is loose and web services consumers and providers send messages to each other using open Internet protocols.

Network transparency refers to a web services capacity to be active anywhere on a network or group of networks without having any impact on its ability to function.

The greatest benefit that web services provide is interoperability at the syntactic level. Web services can be ported to any platform and be written in different programming languages. Similarly, the client accessing the web service can be an application written in a different language and running on a different platform than that of a service itself.

Approaches for Web Service Development

Two of the most widely used approaches for developing web services are Simple Object Access Protocol (SOAP) (Primer, 2006) and Representational State Transfer (REST) architecture style (Hernández and García, 2010). A web service involves three types of roles namely a service consumer, a service provider and an optional service registry.

Figure 2.1 shows the interaction between the service provider, the service consumer and the service registry.

The service providers furnish the services over the web and respond to web service requests. The service consumer consumes the services offered by the service provider.

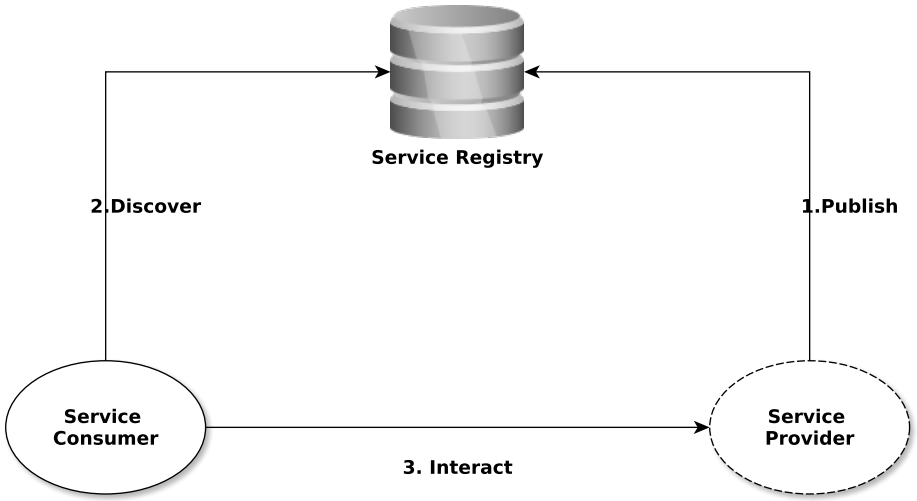

FIGURE 2.1: Relationship of the Service Consumer, Service Provider and Service Registry

In SOAP based web services, the service provider publishes the contract file of the service over the web where a consumer can access it directly or by looking up a service registry. The service consumer usually generates a web service client code from the contract file using the tools offered by the web service framework to interact with the web service (Zur Muehlen et al., 2005).

With REST based web services, there is no formal contract between the service provider and the service Consumer. The service requester needs to know the format of the message, for instance, XML or JavaScript Object Notation (JSON) and operations supported by the service provider. The service provider exposes the set of operations using standard HTTP methods like GET or POST. The service requester invokes one of the methods defined for the resources using the URI over the HTTP protocol (Zur Muehlen et al., 2005).

We agree with (Pautasso et al., 2008) who concluded in their study that the choice of adopting SOAP rather than REST depends on an application's requirements. If the requirement consists of transmitting and receiving simple XML messages, then REST based web services would be appropriate. However, if the requirement consists of various contracts to be defined and negotiated between the provider and consumer such as using a contract file called Web Service Description Language (WSDL) and adhering to various web services specifications such as web service security for enterprise adoption, then SOAP based web services is the right option.

SOAP Based Web Services

Developing SOAP-based web services requires various contracts to be defined and negotiated between the provider and consumer such as using a WSDL for describing the message, adhering to various web service specifications (WS Specifications like WS-Security) and choosing a document literal or Remote Procedure Call (RPC) style for web service communication, as defined in the book (Snell et al., 2002).

A SOAP based web service depends on three interrelated, XML based software standards to function properly:

1. Simple Object Access Protocol (SOAP)-The message format
2. Web Services Description Language (WSDL)-The document that describes exactly what the web service does and how to invoke it
3. Universal Discovery Description and Integration (UDDI)-The directory of web services that are available for use

Together, the three standards combine to give a web service the ability to function,describe itself and be found within a network. While theoretically a web service could function fully using SOAP alone, it needs a WSDL and a UDDI to be effective.

Web service SOAP communication styles

The web service SOAP communication style plays a significant role in communicating SOAP XML messages between the service provider and the service consumer. There exist two types of SOAP message styles, Document literal and RPC (Govindaraju et al., 2004).

The SOAP message styles are defined in a WSDL document as SOAP binding. A SOAP binding can have either an encoded use or a literal use. Encoding implies that the message would be packaged using some format, while literal specifies plain text messages without any wrapping logic.

Document style deals with XML documents as payloads which adhere to well defined contracts,typically created using XML schema definitions. The XML schema format specifies the contract for business messages being exchanged between web service

provider and consumer, which the consumers can call and adhere to. The XML schema defines the request and response message format between the service provider and the service consumer. Document literal style is the preferred way of SOAP based web service communication for achieving interoperability (Ng et al., 2004).

As documented in the SOAP primer, (Gudgin et al., 2003), RPC style on the other hand indicates that the SOAP body contains an XML representation of a method. In order to serialise method parameters into the SOAP message so it can be de-serialised back by any web service implementation, the SOAP specification defines a standard set of encoding rules. As RPC is traditionally used in conjunction with SOAP encoding rules, the combination is referred to as RPC/encoded. One could also have an RPC/literal communication style model where you do not have any encoding formats, but the messages are still limited to RPC method based communication where messages can't be validated as they are not tied to any XML Schema definition.

REST Based Web Services

Roy Fielding in his doctoral thesis, (Fielding, 2000), titled Architectural Styles and the Design of Network-based Software Architectures, wanted to understand why the Web has been so successful. In the thesis, he identifies specific architectural principles that answer the following questions:

1. Why is the Internet so prevalent and ubiquitous
2. What made the Internet scale
3. How could he apply the architecture of the internet to his own applications

This section summarises the conclusions of his thesis. He called these architectural principles Representational State Transfer (REST). REST is identified by the principles of addressable resources, constrained interfaces using HTTP verbs, representation, statelessness and Hypermedia. Below are the listed properties that a REST based application must exhibit:

1. **Addressable resources** Each resource, which is a key abstraction and data in REST, must be addressable via a URI.

2. **A uniform, constrained interface** A small set of well-defined methods to manipulate your resources such as GET, POST, PUT, DELETE, HEAD, TRACE and OPTIONS in the HTTP protocol must be used.

3. **Representation-oriented** Interact with services using representations of that service. A resource referenced by one URI can have different formats. Different platforms need different formats. For example, browsers need HTML, JavaScript needs JSON and a standalone application may need XML (Pautasso et al., 2008).

4. **Communicate in a stateless mode** Stateless means that there is no client session data stored on the server. The server only records and manages the state of the resources it exposes. If there is need for session specific data, it should be held and maintained by the client and transferred to the server with each request as needed.

5. **Hypermedia As The Engine Of Application State (HATEOAS)** Hypermedia is a document centred approach with the added support for embedding links to other services and information within that document format. When first querying for a list of patients, clients do not have to know that they are only getting back a list of 10 patients. The data format can tell them that they did not get a full set. To get the next set, they need to follow a specific link. In this way the data formats drive state transitions.

Understanding REST and RESTful Web Services

The REST architectural style is based on synchronous request and response messages transferred between clients and servers without any of the participating players or intermediaries keeping track of the state of previous sessions.

REST uses nouns and verbs for readability. Resources are identified in requests. The representation of the resource that is sent to the client depends on the request and how the server sends the data.

A REST based Web Service is a service whose interface and accessing mechanism are aligned with the REST principles . The URIs identify the resources. For example,

a RESTful resource for a patient can be identified as http://hospital.org/patient. A resource for a patient identified by patient ID could be http://hospital.org/patient/id/123 . This shows a human-readable URI that is easy to understand and identify.

A client has enough metadata of a resource to modify or delete it as long as it is authorized to do so. To get a resource the client would send an HTTP GET request. To update the resource the client would send am HTTP PUT request. To delete a resource the client would send an HTTP DELETE request. To create a new resource, and for arbitrary processing, the client sends an HTTP POST request.

The GET request retrieves a representation of a resource from a server. The POST request is used to create a resource on the server based on the representation that the client sends. The PUT request is used to update or create a reference to a resource on server. The DELETE request can delete a resource on server. The HEAD requests checks for a resource without retrieving it.

Safety and Idempotence

Two important terms associated with REST are safety and Idempotent.

In the world of REST, a safe method, by definition, is an HTTP method that does not modify the state of the resource on the server. For example, invoking a GET or a HEAD method on the resource URL should never change the resource on the server. PUT method is considered not safe because it usually creates a resource on the server. DELETE method is also considered not safe because it will delete the resource on the server. POST method is not safe since it will change the resource on the server.

Idempotent method is a method that can be called several times without changing the outcome. GET and HEAD methods of HTTP are idempotent because even though the same operation is done many times, the result does not vary. PUT method is idempotent; calling the PUT method several times will not change the result and the resource state is exactly the same. DELETE is idempotent because once the resource is deleted it is gone, and calling the same operation multiple times will not change the outcome. However, POST is not idempotent and calling POST multiple times can have different outcomes.

Media Formats

The response sent by the server could be in XML, JSON, or any other MIME type as long as the server supports the requested format. If the server cannot support the requested MIME type, it will return with a status code of 406 (Not Acceptable).

HATEOAS

When we are developing with RESTful principles in mind, each message should have enough information to let the server understand the purpose of the message and how to process that message, while ensuring visibility and statelessness. This, in REST, is achieved by HATEOAS. According to the Richardson Maturity Model (Fowler, 2010), HATEOAS is considered the final level of REST.

From the human and browser perspective, the Internet is connected together through a series of hyper links embedded within HTML documents. These links enable people to navigate around by clicking through these links within their browsers. Additionally, websites have HTML forms which enable humans to passing information to the server. The browser renders a web page into a format that humans can easily understand. Humans read the web page, fill out information required and submit the form. An HTML form is a self-describing interaction between the client and server.

HATEOAS is the architectural principle that describes linking and form submission. HATEOAS stands for Hypermedia As The Engine Of Application State. The basis of HATEOAS is that data format provides extra information on how to change the state of the application.

The model of application is an engine that moves from one state to another by picking alternative state transitions in current set of representations.

State is where the user is, that is, the current resources. Transition is the instructions on users next steps, that is, links or forms from current resources to others

On the Web, HTML links allow a user to change the state of the browser. When a user is reading a web page, a link tells the reader which possible documents (states) he or she can view next. When a user click a link, the browsers state changes as it visits and renders a new web page. HTML forms provide a way for a user to change the state of a specific resource on the server.

The important point is that data transferred to the client must contain all the necessary information required for the next course of action. This information must be embedded as links in the response. Additionally, a response contains standard status codes like 200 for successfully operation or 404 for a resource that has not been found on the server.

The following code represents a Patient object.

```
class Patient {
    String name;
    int age;
}
```

A simple JSON presentation is traditionally rendered as:

```
{
    "name" : "Peter"
    "age"  :  20
}
```

The listing above just shows the Patient data with name Peter and age 20, but the data contains nothing about its relevant links about what next course of action to be done by the receiver.

A HATEOAS-based response would look like this:

```
{
    "name": "Peter",
    "age" :  20,
    "links": [ {
        "rel": "self",
        "href": "http://localhost:8080/patient/1"
    },
    {
        "rel": "address",
        "href": "http://localhost:8080/patient/1/address"
    }
    ]
}
```

This response not only has the patients's name, but also includes the self-linking URL where that person is located on the server and the link to where the patient's address resource is located. The variable **rel** defines the relation. In this case, the first one is a self-referencing hyper link and the second one is an address-referencing hyper link. These links can be added as needed.

It is possible to build more complex relationships. With HATEOAS, the output makes it easy to glean how to interact with the service without looking up a specification or other external document. Below is another example of a HATEOAS output in JSON

```
{
"content": [ {
    "description": "HIV Test Appoitment",
    "name": "Chanda Phiri",
    "links": [ {
        "rel": "self",
        "href": "http://localhost:8080/appointment/1"
    } ],
    "attributes": {
        "visiType": "repeat"
    }
}, {
    "description": "Social Worker Appointment",
    "name": "Mary Jane",
    "links": [ {
        "rel": "self",
        "href": "http://localhost:8080/appointment/3"
    } ],
    "attributes": {
        "visitType": "new"
    }
} ],
"links": [ {
    "rel": "appointment.search",
    "href": "http://localhost:8080/appointment/search"
} ]
}
```

In the listing above, not only are the appointment items and their details shown, but the URL for each resource is shown. No session is required in this interaction as the client is given all the information needed to interact with the service or initiate the new request. If any of the links is broken, the client will be given an appropriate error code.

Addressing perceived Challenges of REST Based Web Services

The statelessness property advocated for REST based Web Services and the uniform interface constraint imposed poses a number of challenges when designing architectures that involve operations that require application session state.

Such operations, among many others, include making copies, taking snapshots, moving and merging, processing batch requests, supporting transactions, security and dealing with failure.

These issues are not necessary addressed in the same way that tradition distributed systems solve by creating wrapper protocols. These issues are solved by designing things in a different way , while sticking to REST principles because there is no luxury of extending the HTTP protocol available to the architect. This section will address some of the work the designs around these issues.

For example, let us consider moving a resource on the server to a different location. Undoubtedly, this operation in a traditional distributed system will require session involvement. However, what a move operation means is completely application specific. It could mean copying a resource to a different location on the same or a different server and deleting the original. Alternatively, it could also mean changing the state of a resource without changing its location. In either case, to maintain loose coupling, the client should not concern itself about what a move means and how the server implements it. The client just needs to initiate the operation.

Another example is security, which require session to be active. This is also designed differently by requiring that a client sends an authentication token every time they send in a request to the server.

We shall look at two of the main issues in detail namely Maintaining Application State and Transactions

Maintaining Application State

The recommendation when using REST architectural style is to keep the **application state** on the client. Application state is the state that the server needs to maintain between each request for each client without relying on in-memory sessions on servers.To achieve this, application state is encoded into URIs included into representations via links. The clients use these URIs to interact with resources.

Let us consider a simplified use case of a two steps process of making an appointment to see a Doctor. In the first step, the patient (could be a system) submits a request asking for the list of available Doctors and the server returns a list of Doctors with their available times. In the second step, the patient submits a request to book an appointment. In this example, the application state is the list of appointments. The server needs to know the Doctor and available time from the first step so that it can issue an appointment ticket based on that list in the second request.

Since HTTP is a stateless protocol, each request is independent of any previous request. However, interactive applications often require clients to follow a sequence of steps in a particular order. This forces servers to temporarily store each clients current position in those sequences outside the protocol. In order to strike a balance between reliability, network performance, and scalability, it is better to maintain application state within links in representations of resources.

If the state is large or cannot be transported to clients for security or privacy reasons, it is better for the server to store state data in a data store and encode its primary key in the URI. When the client makes a request to to the server using encoded URI, the server restores the application state using the encoded key. Stored state can be replicated and cleaned up at some point.

Transactions

One of the key challenges of using RESTful web services is how to deal with transactions. The following two scenarios prompts the need for transactions:

- A client goes through a sequence of steps with a server in a flow. The client would like to cancel the flow and undo all the changes done to the data in that flow.

- A client interacts with a number of servers in a sequence to implement an application flow, and the client either wants to revert any resulting state changes or wants to record them permanently.

The Stateless nature of HTTP Protocol does not support this type of use case and a shift in thinking is required to meet the above use case.

There are a number of types of transaction models notably, Short-lived atomic transactions to guarantee properties such as atomicity, consistency, isolation and durability (ACID), and long-running transactions where changes happen over a period of time.

For example, creating a patient record in a data store may happen under an atomic transaction, whereas buying a drugs or making a hospital reservation may need to happen over a long running transaction.

In the case of HTTP, which is a synchronous protocol, each request provides a sphere of control for atomic changes. An HTTP request either succeeds or fails. There is no scope for partial failures in HTTP. Therefore, support for atomic operation can be provided by submitting changes to a resource and let that resource attempt to commit those changes by using optimistic concurrency control. The server can provide guarantees such as atomicity, consistency, isolation and durability as appropriate by delegating the task of committing changes to a transactional server data store.

Taking a classic example of moving money from one account to another in a transaction, a resource can be used to atomically update two bank account resources within a single request shown in listing below.

```
1  # Request
2  POST /transfer;t=ytr232544675tuy87766786t HTTP/1.1
3  Host: bank.com
4  Content-Type: application/xml;charset=UTF-8
5
6  <transfer>
7      <source>urn:bank:com:currentaccount:1</source>
8      <target>urn:bank:com:savingsaccount:2</target>
9      <currency>ZAR</currency>
10     <amount>10000.00</amount>
11     <note>Saving For the Rainy Day</note>
```

```
12  </transfer>
```

And the backend Server would respond with

```
1   # Response
2   HTTP/1.1 201 Created
3   Content-Type: application/xml;charset=UTF-8
4   Location: http://www.bank.com/transactions/1
5
6   <transfer xmlns:atom="http://www.w3.org/2005/Atom">
7      <source>urn:bank:com:currentaccount:1</source>
8      <target>urn:bank:com:savingsaccount:2</target>
9      <atom:link href="http://www.bank.com/transactions/1" rel=↩
       "self"/>
10     <currency>ZAR</currency>
11     <amount>10000.00</amount>
12     <note>Saving For the Rainy Day</note>
13  </transfer>
```

To the client, this is an atomic activity. The client is decoupled from the details of how the server implements the atomic activity. The server handles concurrency control either by checking conditional headers in the HTTP methods.

Long running transactions, such as two-phase commit, can be supported by introducing links to compensate for actions. For example, the server can provide a link to cancel or update an action in the example of a reservation resource shown below.

```
1
2   # A Hotel Reservation
3   GET /reservations/2 HTTP/1.1
4   Host: www.hotel.com
5   # Response
6   HTTP/1.1 200 OK
7   Content-Type: application/xml;charset=UTF-8
8
9   <reservation xmlns:atom="http://www.w3.org/2005/Atom">
10    <reservation-id>2</reservation-id>
11    <atom:link rel="self" href="http://www.hotel.com/↩
       reservations/2"/>
```

```
12    <atom:link rel="http://www.hotel.com/rels/cancel"
13                href="http://www.hotel.com/reservations/2/↩
       cancel"/>
14    <!-- details of the appointment -->
15      ...
16  </reservation>
```

The client can use the link with the relation **http://www.hotel.com/rels/cancel** to cancel the reservation and perform the necessary compensating actions such as cancelling the travel segment, deducting cancellation fees and refunding the money.

Managing transactions this way keeps clients decoupled, keeps interactions stateless and guarantees atomicity within each HTTP request.

REST Vs SOAP comparison

SOAP based web services have been effectively used within SOA space as a solution for many heterogeneous interconnectivity problems because it was designed to take advantage of many different types of data transport layers including synchronous HTTP/HTTPS, asynchronous. The initial solution for the work in this research was based on SOAP based web services.

However, it was found in the first case that changing services that use SOAP often meant changing the code on the client. Generating SOAP client code from WSDLs and XSDs often broke the existing client code. Re-coding the SOAP interfaces to meet the clients interfaces was time-consuming and error-prone.

A study on the comparison of REST and SOAP based web services by (Wagh and Thool, 2012), revealed most of the challenges faced in the early development of the case study I. Although the study is focused on mobile phone communications, the constraints imposed by wireless communication infrastructure are similar to the challenges faced in most resource constrained environments.

One of the conclusion the study made was that REST based framework was suitable for handheld, resource constrained mobile device and wireless network. The Study also

found that the level of resource consumption is less in REST based web services as compared to SOAP based web services.

2.2.5 Enterprise Application Integration

Enterprise Application Integration (EAI) as described by (Linthicum, 2000) in their book is the application of technology defined as the integration of data and services between systems. The problems it addresses are boundless and the solutions almost as myriad. Despite having been building integration solutions for decades, it is only now that developers have been able to identify the best practices and categorise them.

The patterns of modern day EAI are usually attributed to Enterprise Integration Patterns (Hohpe and Woolf, 2003) that categorises and explains many of the integration patterns common to integration solutions today. (Hohpe and Woolf, 2003) list four types of integration styles:

1. **File Transfer:** Two systems produce files whose payload is the message to be processed by the other system. System exchange messages using the common File Transfer Protocol (FTP) or Secure Shell (SSH) protocol.
2. **Shared Database:** Two systems query the same database for data to be passed.
3. **Remote Procedure Call:** Each of the two systems expose services that the other can call.
4. **Messaging:** Two systems connect to a common messaging system and exchange data and invoke behaviour using messages.

These styles are disparate because no one solution will work all the time. This has given rise to a whole field of middleware that seek to enable solutions based on these patterns, typically called an Enterprise Service Bus (ESB) as described in the conference paper by (Schmidt et al., 2005). An ESB is the ultimate mediator which knows how to talk all languages over all protocols and mediate messages being passed.

There are many patterns for EAI and just as many protocols that need to be dealt with. Also, there are several things that are common to most mediating systems. Some of the most important ones are:

1. Routing: The ability to route messages on conditional logic or configured routing rules.

2. Messaging enhancement and transformation: Transforming the message's payload from one type to another

3. Mediation: To provide adaptors and service mapping between different messages.

2.2.6 Messaging

Web services are simply the evolution of the techniques for moving messages between computers in distributed environments. Web services have brought a standardised path to accomplish these goals. Two systems can communicate through a variety of methods ranging from the internet to dedicated wide area networks (WANs) or local area networks (LANs).

Messaging is at the core of virtually every issue and solution related to managing distributed computing as shown by (Bernstein, 1996). Whether it is Electronic Data Interchange (EDI), Enterprise Application Integration (EAI) or any one of a dozen other ways in which IT vendors have attempted to address challenges in distributed computing, the core issue is almost always the same: transmission, receipt of and response to messages between the computers involved (Schmidt et al., 2005).

Most applications rarely live in isolation and all integration solutions have to deal with a few fundamental challenges identified by (Halevy et al., 2005):

1. Computer networks are unreliable. Integration solutions have to transport data from one computer to another across networks. Compared to a process running on a single system, distributed computing has to be prepared to deal with a much larger set of possible problems. Often times, two systems to be integrated are separated by physical distance or location and data between them has to travel through various media which can fail and cause delays and disruptions.

2. Networks are slow. Sending data across a network is slower than making a local method call. Designing a widely distributed solution with a single application assumptions could have disastrous performance implications.

3. Any two applications are different. Integration solutions need to transmit information between systems that use different programming languages, operating platforms and data formats. An integration solution needs to be able to interface with all these different technologies.

4. Change is inevitable. Applications change over time. An integration solution has to keep pace with changes in the applications it connects. An integration solution needs to minimise the dependencies from one system to another by using loose coupling between applications.

To solve the above problems, we look at two sources of information chronicled by (Ruh et al., 2000) and Hohpe and Woolf (2003) in their books. Both identify two main approaches to deal with these problems.

The first solution is to abstract the remote communication as local and build applications based on that assumption. The second solution is to use a messaging systems that can guarantee delivery of message communication.

Many integration approaches are aimed at making remote communication simple by packaging a remote data exchange into the same semantics as a local method call. This strategy resulted in the notion of a Remote Procedure Call (RPC) or Remote Method Invocation (RMI) supported by many popular frameworks and platforms such as the Common Object Request Broker Architecture (CORBA) , Microsoft Distributed Component Object Model (DCOM), .NET Remoting, Java RMI and most recently, RPC-style web services.

(Kendall et al., 1994) describe the challenges in their paper in great detail. The challenge that these approaches face lies in the fact that remote communication invalidates many of the assumptions that a local method call is based on. As a result, abstracting the remote communication into the simple semantics of a method call can lead to problems because of the many assumption made by the communicating parties.

The more assumptions two parties make about each other and the common protocol, the more efficient the communication can be, but the less tolerant the solution is to interruptions or changes because the parties are tightly coupled to each other. (Kendall

et al., 1994) clearly stated this as far back as 1994 that objects that interact in a distributed system need to be dealt with in ways that are intrinsically different from objects that interact in a single address space.

The second solution is to reduce assumptions communication parties make about each other when they exchange information. This in turn guarantees loose coupling. Messaging enables data or commands to be sent across the network in such a way that the caller sends the information and then goes on to do other work while the information is transmitted by the messaging system. Optionally, the caller can later be notified of the result through a callback. This type of communication is called asynchronous. As opposed to synchronous message systems, players in an asynchronous messaging system do not have to wait for a response from the recipient because they can rely on the messaging system to ensure delivery.

This is a vital ingredient in loosely coupled systems such as REST based web services because it allows participants to communicate reliably even if one of the parties is temporarily offline, busy or unreachable. Asynchronous messaging systems are also vastly more scalable than those that rely on direct connections, such as remote procedure calls (Fall, 2004).

Messaging system

Messaging capabilities are typically provided by a separate software system called a messaging system or Message Oriented Middleware (MOM) (Curry, 2004). The messaging system coordinates and manages the sending and receiving of messages. The primary purpose of a messaging system is to move messages from the senders computer to the receivers computer in a reliable fashion.

The reason a messaging system is needed to move messages from one computer to another is that computers and the networks that connect them are inherently unreliable. A messaging system overcomes these limitations by repeatedly trying to transmit the message until it succeeds. Under ideal circumstances, the message is transmitted successfully on the first try.

Basic Messaging Concepts

Like most technologies, messaging involves certain basic concepts. From the (Banavar et al., 1999), these basic messaging concepts are:

1. **Channels or Queues** – Messaging applications transmit data through a Message Channel, a virtual pipe that connects a sender to a receiver.
2. **Messages** – A Message is an atomic packet of data that can be transmitted on a channel.
3. **Multi-Step Delivery** – The process of changing a message into an appropriate format before it is delivered to the final destination.
4. **Routing** – Process of deciding where the message is to be routed based on a number of pre-determined conditions.
5. **Transformation** – This refers to the filters in the middle that acts like a translator for changing the format of a message from one format to another.
6. **End Point** – This is a bridge piece of software which has a set of coordinated message points that enable the application to send and receive messages.
7. **Producer or Sender** – Is a program that sends a message by writing the message to a channel.
8. **Consumer or Receiver** – Is a program that receives a message by reading (and deleting) a message from a channel.

Challenges of Asynchronous Messaging

Asynchronous messaging is not the silver bullet of integration as shown by the work done by (Geihs, 2001). While resolving many of the challenges of integrating disparate systems in an elegant way, asynchronous messaging introduces new challenges. Some of these challenges are inherent in the asynchronous model while others vary with the specific implementation of a messaging system. Some of the challenges are listed below from (Hohpe and Woolf, 2003).

1. The Event-Driven programming model used is complex. Application logic can no longer be coded in a single method that invokes other methods, but the logic is split up into a number of event handlers that respond to incoming messages. Such a system is more complex and harder to develop and debug.

2. Sequence message delivery is not guaranteed and the developer has to use his or her judgement to determining where asynchronous messaging will have to be used.

3. Messaging systems do add some overhead to communication in terms of performance. It takes effort to make data into a message, send it, receive it and process it. Data size can be an issue.

2.2.7 Service Oriented Architectures

Software Architecture

The definition used in defining architecture by (Korpela et al., 2005) will be used in this study. Architecture is the set of decisions the software architect makes. Software architecture is commonly defined in terms of structural elements and relationships. Structural elements are identified and assigned responsibilities that client elements interact with through contracted interfaces.

An architecture is not a simple flat view of the component topology. However, an architecture diagram showing the components and relationships among them is a central thinking and communicating tool for the architects and development teams. An architecture needs to include:

1. Meta-architecture: The architectural vision, style, principles, key communication and control mechanisms, and concepts that guide the team of architects in the creation of the architecture.

2. Architectural views: Just as building architectures are best envisioned in terms of a number of complementary views or models, so too are software architectures. For example:

(a) Structural views help document and communicate the architecture in terms of the components and their relationships. They are useful in assessing architectural qualities like extensibility.

(b) Behavioural views are useful for thinking through the way components interact to accomplish their assigned responsibilities and evaluating the impact of what-if scenarios on the architecture. Behavioural models are especially useful in assessing runtime qualities such as performance and security.

(c) Execution views help in evaluating physical distribution options, documentation and communicating decisions.

In creating these views, attention is paid to :

1. Architectural patterns: Structural patterns such as layers, client/server and mechanisms such as brokers and bridges.
2. Key architectural design principles including abstraction, separation of concerns, postponing decisions and simplicity with related techniques such as interface hiding and encapsulation.
3. Design of architectural mechanisms, where mechanisms deal with the interaction of components to achieve specific system capabilities.
4. System decomposition principles and good interface design.

Enterprise Architecture

For Enterprise Architecture (EA), the bulk of the material comes from the book by (McGovern et al., 2003). EA is the total picture of an organisation's IT systems. It shows how systems relate to each other and how they drive the processes. The discipline of creating EAs is known as enterprise architecture planning (EAP).

Traditionally, EAs have been inflexible, confined by the structures of network protocols, hardware platforms and programming languages. It is this tight coupling of systems in traditional EAs that has rendered them costly and time consuming to modify because Enterprise Architects have difficulty keeping pace with changing business processes.

Service Oriented Architecture

The Service Oriented Architecture (SOA) is an approach to EA where each major element is exposed as a service. The result is a distributed computing environment with a high level of syntactic interoperability between systems. SOA enables the enterprise architect to combine and recombine software elements without the need to spend substantial amounts of time or money, assuming it has been implemented intelligently (Hashimi, 2003). Because of flexibility, easy-to-change and economical nature, SOA gives people the ability to modify their processes without the kind of IT restrictions that have hampered such changes in the past.

The idea of SOA is that application developers design their systems as a set of reusable,decoupled, distributed services. Since these services are published on the network,conceptually, it should be easier to compose larger and more complex systems. SOA is more open and provides greater enabling of interoperation without reliance on proprietary vendor platforms.

Industry efforts such as the OASIS group deals with defining the terms and conditions of using and deploying SOA based systems. The objective of the OASIS group is not only to provide a simple approach to the service question in service architecture, but also a mechanism for planning, managing and delivering projects using SOA techniques.

There is a huge number of definitions as to what a service is and what service oriented architecture means. Many of these definitions come from vendors and are linked to the products they have to sell.

However, there has been a successful attempt inside OASIS group to define a reference model for SOA which has resulted is in a public draft. The reference model does not tell you how to do SOA, but just helps you to determine what good SOA looks like. The reference model defines the various terms that are used in SOA. It is strongly advised to take the reference model as a basis for a project or enterprise definition of SOA. The model has the advantage of being vendor, technology and business neutral. Additionally, it comes from a recognised standards body.

According to the OASIS group, SOA is a paradigm for organising and utilising distributed capabilities that may be under the control of different ownership domains. It

provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectation (OASIS, n.d.).

A key factor in the SOA reference model is that it is not limited to just technology, a common failing in most other views of SOA, but can be applied to all facets of an organisation, including IT, infrastructure and the pure business elements themselves.

This is critical when looking to model a service architecture. If an approach or definition is limited to technology then it will fail as it will only cover those elements that are in technology today, and not those that drive those technologies, and which may become automated in future. SOA has to be about all the services in the business.

In the health care context, Health institutions, which represent ownership domain, create capabilities to solve or support a solution for the problems they face in the course of running their institution. Capabilities can be locally owned by the institution or be under the control of another institution like a specialist laboratory. These capabilities, located at one or various places, can be combined to provide what is known as a service. In SOA, services are essentially the mechanism by which needs and capabilities are brought together (OASIS, n.d.).

SOA goals, as a general principal, have been around for many years. A number of technologies like Common Object Request Broker Architecture (CORBA), Distributed Component Object Model (DCOM) and Java Remote Method Invocation (RMI) have attempted to realise them. Although it is clear that these technologies were promising, they failed because of vendors adapting proprietary implementation of the ideas. Further, these technologies (CORBA, DCOM, RMI, etc) depend on specific binary protocols that internet firewalls, by default, are configured to block. They tend mostly to work well within a Local Area Network (LAN). Deploying these technologies over the public internet is a non-trivial mission. It needs special firewalls or gateways.

Web services are one effective way to create SOA because the infrastructure and standards that support this technology are finally in place to make web services based SOA practical. Please note that SOA is not equal to web services. It can be implemented using other technologies with much difficulty. The big motivation for web services based

SOA is that the infrastructure is now in place. We now have the infrastructure that gives us new ways to create, manage, share and secure applications.

By creating connections among disparate applications, not previously possible due to their inherent incompatibility, the SOA approach can help institutions increase the flexibility of their clinical processes, strengthen their underlying IT infrastructure, retain and reuse the IT services they have already deployed.

These connections can allow a complete clinical process to be linked to the exact IT components needed to execute the processes. Further, these connections can be mixed, matched and re-used to address problems unique to individual institutions. Since SOA based components are based on industry standards, services connected in this way can also be shared with other various institutions.

Additionally, within an institution, using SOA to connect information from disparate medical systems can give caregivers access to multiple applications, including lab reports, medical imaging, scheduling and other critical information about a patient.

The OASIS SOA reference model talks about the execution context, which is the “thing” that enables the consumer and producer to communicate. It is at this level that technology is important as the enabler between these two parties. The execution context has nothing to do with the value ascribed to that invocation nor to the reasons for that invocation happening.

A service architecture broadly follows a four step process or more accurately delivers three stages of the process and provides the direction for the fourth.

1. What: Defining the scope of services. This is about determining what the services actually are.
2. Who: Who are the external actors that drive the services or with which the services interact.
3. Why: Identifying why one service talks to another and why external actors interact with the services.
4. How: The details about the processes that co-ordinate the services and also the details on how a service itself will be implemented.

2.2.8 Service Oriented Architecture Approaches

There are three main approaches when it comes to building service oriented architectures. First is the top down where central authority decides everything and the implementation teams adopts them. Second is the bottom up where central authority provides a directory and implementation teams make whatever services they want. The implementation teams go to the directory to find services they can use. Lastly, the hybrid approach which is a mixture of both the top down and the bottom up approach.

The Top Down SOA Model

SOA's aim is to solve integration problems by shifting internal application development practice towards the creation of re-usable components called services which can easily talk to each other.

In a top down approach, an institution first makes a comprehensive map of the actual functions they need from their infrastructure. The map includes information such as the task the silo programs are meant to address, how several silos relate to each other and what kind of data formats and protocols have to interoperate. Next, the institution determines how each of these functions could be expressed as a collection of services (Meijler et al., 2006).

From this assessment, the organisation is able to identify those services that are common to every application and build them in such a way that the same service would be re-usable in every application.

Once these services have been created, the various applications that the company had been using before could be re-created with minimal duplicated code by using these common parts (Woodley and Gagnon, 2005). As an additional piece of the plan, any functions specific to a new application would also be created as new services and made available for re-use by any later applications that might require them.

Therefore, SOA creates an ecosystem of actively updated components of business logic that could be quickly linked with minimal amounts of new code to create ad-hoc programs to handle any business need, no matter how specific.

While this approach for implementing SOA looks great on paper, it has difficulties when put into action (Meijler et al., 2006). It is slow and difficult to follow the top down approach despite it being functional. Most of the problems are caused by inaccurate assumptions about organisational behaviour that are included in every top down adoption plan.

Top down approach requires a lot of strategy formulation, a strong commitment from management and everybody involved in an organisation to come together to define candidates for services and requirements for these services. It demands more of an initial investment because it introduces an up front analysis stage focused on the creation of the service inventory blueprint.

Service candidates are individually defined as part of this blueprint so as to ensure that subsequent service designs will be highly normalised, standardised and aligned.

With everybody involved, the inevitable consequence is the slow process at reaching consensus. This process can even be slower than the rate at which institution's requirements change. Top down is particularly hard to follow for big organisations. It is more likely to succeed in small organisations.

Bottom Up SOA Model

The idea of Bottom Up approach is that many services can be developed without much central control as long as they are in a directory. Developers would look in the directory to find the services they need and adopt the best or least expensive.

The premise for the bottom up approach is that an IT organisation builds a service oriented application solely because doing so makes sense for them. The architects and developers who create this application know that it will be accessed by diverse clients or other applications. So, making it service oriented from the start will make their lives easier.

The IT organisation then builds another service oriented application and so on (Meijler et al., 2006). Whenever required, an existing application is wrapped with services to let it participate in this new world.

The problem with bottom up is that you end up with systemic effects not seen by individual contributors. The result of this is that it makes it difficult to easily see the broader

view of the organisation (Twidale and Floyd, 2008). It is also difficult to get even contributions from the individual departments as some contribute more or consume or or even totally just ignore the repository. Also the repository may suffer the local specific tag as individual contributors just do so to solve their individual problems. As a result, the repository may end up becoming a white Elephant not worth producing.

Measuring SOA success by the number of services or encouraging the creation of services without any form of central understanding of what the services should contain is not perceived as success by the supporters of the top bottom approach.

The Hybrid Approach

This is a mixture of both the top down and the bottom up approach. (Twidale and Floyd, 2008) study documents the merits of this approach. The premise is that central group provides key elements of the interface,including numbering schemes, message exchange patterns, standard communication mechanisms and monitoring infrastructure. The central group also encourages the development teams to use it to build services that can be shared. The bottom up is usually driven with the top down structure where the key contact interfaces of the organisations are made available in the repository for team to use in their bottom up approaches.

2.2.9 Enterprise Service Bus

In this section, the Enterprise Service Bus (ESB) is looked at because it takes a different approach in addressing a number of issues raised in this work.

There are a number of perspectives from which ESBs can be looked at. First, as a pattern that is implemented with several different products, each excelling in its own domain like routing, transformation, security and orchestration. Secondly, as a product offering that provides integration functionality, a developer tool set and a management environment.

An ESB is as an important part of a service-oriented architecture (SOA). From the SOA perspective, an ESB is used as an integration platform that enables existing IT assets and applications to be exposed as services. Because the ESB is based on open standards, the proprietary technology of legacy applications can be exposed as

services based on open and modern technologies like web services and messaging. This view is well documented in the work done by (Menge, 2007).

This work did not look at ESBs as a product to be deployed, but rather as a pattern to be implemented. Most Open Source ESBs products at the beginning of this work were not lightweight and required a lot of resources to manage and deploy. They could not be embedded easily in any new products. Furthermore, commitment to a particular vendor implementation is bound to force anybody using the integration library to have to stick to the implementation that is shipped with the library.

During one of the case studies where ServiceMix ESB (ServiceMix, 2007), an open source ESB, was involved, it was found that the transformation of data was creating significant network traffic because messages passing through the integration hub are were routed in real time and communication channel had to stay open for data transformation and lookups. The blocking nature of all the available open Source ESBs created problems in delivering responsive high throughput of messages.

Therefore, decision was made to go with the ESB pattern and use a lightweight library like Apache Camel for implementation.

2.3 Health Informatics

Health Informatics (HI) is one of the fastest growing areas in the health care industry as shown by the paper done by (Wilson and Tulu, 2010). Its importance is underscored by the fact that it provides information that help in making decisions. Good information leads to good decisions. Health care management, planning and policy all need good information.

There are a number of definitions for HI. In this work, the used definition is proposed by (Tan and Payton, 2009) after an extensive research on the term “Health Informatics”. They define Health Informatics (HI) as follows:

“ Health Informatics is the mature discipline which fulfils its pivotal role in the collection, storage, retrieval, synthesis, communication and pertinent use of clinical knowledge discovery. The approaches of health Informatics go further beyond the application of computer technology and problems focused applications in

the health care and optimistically touch relevant aspects of fields like education, research, practice and science. In parallel, it catalyses the vertical and horizontal diffusion of communication, technology, computer science and information science. The nomenclature of the burgeoning field of health informatics concentrates on paving its focus towards the promotion of health care delivery and betterment of patient care."

From the definition it can be seen that HI forms as the meeting point of information science, computer science and health care, a crossroads where information, IT and health care knowledge meet.

Health informatics applies technology and information to enhance health care delivery, support bio medical research and foster education of health professionals and the public (Sullivan, 2001).

This study largely focuses on how health data and knowledge are collected, stored, processed, communicated and used to support the process of health care delivery to clients, providers, administrators and organisations involved in health care delivery. It deals with the resources, devices and methods required to optimise the acquisition, storage, retrieval and sharing of information in health care industry.

From this, it can be seen that the key elements needed to deal with are acquisition, storage, communication, manipulation and display. It is these aspects of data that makes interoperability of health care information a challenge.

The next section looks at the historical background of how this data has been captured stored and shared.

2.3.1 Health Information Systems Standards

We agree with the conclusion made by (Hovenga et al., 1996) that a standard electronic health record (EHR) and an interoperable national health information infrastructure require the use of uniform health information standards, including a common medical language. Data must be collected and maintained in a standardised format using uniform definitions in order to link it within an EHR system or share it between systems.

The lack of standards has been a key barrier to electronic connectivity in health care industry (Hersh, 2004).

Together, standard clinical terminologies and classifications represent a common medical language which allow clinical data to be effectively utilised and shared between EHR systems. Therefore, standard clinical terminologies and classifications, with maps to link them, must be incorporated into EHR systems to achieve systems interoperability (Chute, 2000).

2.3.2 Complexity of Clinical Data

There are two aspects to consider when looking at clinical data. The consensus on the clinical domain model and the type of attributes that go into a domain model. The domain model definition used by (Evans, 2004) is used here. The domain model captures the things that exist and events that transpire in a particular environment. For example, in a restaurant domain one would find waiters and waitress where as you would find nurses and doctors in a hospital domain.

The critical thing in the health domain is the data captured and produced. The format in which this data is captured matters a great deal. There are all types of data formats captured both in paper systems and electronic systems. To ease this confusion, le us turn to the paper done by (Chute, 2000) where the evolution of health terminology is chronicled.

According to the paper, several coding schemes have been developed and theses schemes fall into two categories called classifications and nomenclatures. The paper goes on to say Clinical classifications derive from epidemiology and health information management.

Classification systems such as ICD-9-CM, ICD-10-CM and ICD-10-PCS group similar diseases, procedures and organise related entities for easy retrieval. Classification systems allow granular clinical concepts captured by a terminology to be aggregated into manageable categories for secondary data purposes. They are typically used for external reporting requirements or other uses where data aggregation is needed, like measuring the quality of care, monitoring resource utilisation or processing claims for reimbursement in private institutions.

Classification systems are not intended or designed for the primary documentation of clinical care. They are inadequate in a reference terminology role because they lack granularity and fail to define individual clinical concepts and their relationships (Chute et al., 1996). Yet they are the most common source of clinical data, readily available as a byproduct of the health care reimbursement process by private clinics or hospitals.

Clinical nomenclatures, which are derived from health informatics, are standardised terms. Their synonyms record patient findings, circumstances, events and interventions with sufficient detail in order to support clinical care, decision support, research outcomes and quality improvement. They can be efficiently mapped to broader classifications for administrative, regulatory, oversight and fiscal requirements (Chute, 2000). They are expressed in natural language and codify the clinical information captured in an EHR during the course of patient care. Common examples include Systematised Nomenclature of Medicine–Clinical Terms (SNOMED-CT) and Logical Observation Identifiers, Names, and Codes (LOINC).

SNOMED CT is a comprehensive clinical terminology designed for use in electronic, not paper-based, health record systems. The number of terms and level of detail in a reference terminology cannot be effectively managed without automation. LOINC is a universal standard for identifying medical laboratory observations.

Reference terminologies are inadequate for serving the secondary purposes for which classification systems are used because of their immense size, considerable granularity, complex hierarchies and lack of reporting rules.

Neither a clinical terminology nor a classification can, by itself, serve all of the purposes for which health information is currently used or will be used in the future. Terminologies and classifications are designed for distinctly different purposes and satisfy diverse user data requirements. Multiple terminologies as well as classification systems are necessary to capture and effectively use the breadth and depth of clinical data in an EHR (Chute et al., 1996).

2.3.3 Interoperability of Health Information Systems

According to the National Alliance for Health Information Technology (NAHIT, USA) interoperability in health care is defined as the ability of different information technology

systems and software applications to communicate, exchange data accurately, effectively, consistently and use the information that has been exchanged.

An effective health information systems infrastructure requires collaboration among systems. Each system in the infrastructure requires information from one or more systems. If the information flows among systems is standardised, there will be no need to dictate the functions of the individual applications.

The study conducted by the HL7 Interoperability Working Group, a part of the Electronic Health Record Technical Committee (Gibbons et al., 2007), showed that there were three definitions of interoperability in the health care industry: Syntactic, semantic and process interoperability.

Syntactic or technical interoperability involves communicating data between different applications and systems over a physical network. Meaning of the communicated data is not significant.

Semantic interoperability is the ability to not only exchange data, but also understand the meaning of the exchanged data. Semantic interoperability is about the meaningful exchange of information in association with its context.

Process interoperability is the ability to exchange business work flow processes between systems. It is largely to do with the design and implementation of human work processes that can be exchanged in a meaningful way.

2.3.4 Analysis of Electronic Healthcare Records Standards

The Electronic Healthcare Record (EHR), which includes information such as observations, laboratory tests, diagnostic imaging reports, treatments, therapies, drugs administered, patient identifying information, legal permissions and allergies, is defined by (Iakovidis, 1998) as “Digitally stored health care information about an individuals lifetime with the purpose of supporting continuity of care, education and research, while ensuring confidentiality at all times”.

This information is stored in all kinds of proprietary formats ranging from electronic formats to paper based systems. This has resulted in interoperability problems faced in the health care informatics domain. Making EHRs interoperable has been the centre

focus of many standards bodies worldwide. The goal has been to facilitate the retrieval and processing of clinical information about a patient from different systems.

There are several studies that have been done on this subject of surveying the electronic health standards used in information systems currently in existence. The most comprehensive and most cited work is that done by (Eichelberg et al., 2005). In their paper, they presented a survey of the most relevant electronic healthcare record standards based on the following main questions

1. The level of interoperability support: Does the EHR provide structured content suitable for automated processing? Does it specify content distribution rules?
2. Functionality: Does the standard allow for an explicit retrieval of records (or parts thereof) for a specific patient, based on an incoming request? Can it contain multimedia data? What kind of security mechanisms are supported for accessing healthcare records?
3. Complementary: Since not all the standards provide all the necessary features, is it possible to combine them in a complementary way? Do the standard initiatives affect one another?
4. Market relevance: Is the standard accepted in the marketplace? Are there commercial implementations available or any signs of uptake by industry?

The seven EHR standards evaluated included several of those under development such as the

1. Health Level 7 (HL7) Clinical Document Architecture (CDA).
2. CEN/TC 251 AND ENV/EN 13606 EHRCOM.
3. EHR/openEHR initiative.
4. Integrating the Healthcare Enterprise (IHE) Cross-Enterprise Document Sharing (XDS) integration profile.
5. Integrating the Healthcare Enterprise (IHE) Retrieve Information for Display (RID).

6. Digital Imaging and Communications in Medicine (DICOM) Structured Reporting (SR).

7. The Medical Markup Language (MML).

Health Level 7 (HL7) Clinical Document Architecture (CDA)

HL7 Clinical Document Architecture (CDA) (Dolin et al., 2006) defines the structure and semantics of medical documents for the purpose of exchange. CDA documents are encoded in Extensible Markup Language (XML). They derive their meaning from the HL7 Reference Information Model (RIM) and use the HL7 Version 3 Data Types which are part of the HL7 RIM (Quinn, 1998).

CEN/TC 251 AND ENV/EN 13606 EHRCom

CEN/TC 251 (CEN, 2002*a*) is the technical committee on Health Informatics of the European Committee for Standardisation whose mission is to achieve compatibility and interoperability between independent health systems and to enable modularity by means of standardisation.

The CEN ENV 13606 Electronic Healthcare Record Communication (EHRCom) (CEN, 2002*b*) is a message-based standard for the exchange of electronic health care records. The standard defines an EHR information model, a list of machine-readable domain terms that can be used to structure EHR content, a method of specifying distribution rules. The rules under which certain EHR content may be shared with other systems and, finally, request and response messages that allow systems to exchange subsets of an EHR. Rather than specify a complete EHR system, the ENV 13606 just focuses on the interfaces relevant for communication between EHR systems.

EHR/openEHR initiative

The GEHR/openEHR initiative was started in 1992 as an EU research project called Good European Health Record, which later continued under the name Good Electronic

Health Record with strong participation from Australia. Currently, it is maintained by the openEHR Foundation (Kalra et al., 2005), a non-profit organisation.

The GEHR/openEHR standard introduced the concept of archetype (Garde et al., 2007) which uses a two-level methodology to model the EHR structure. In the first level, a generic reference model with a few classes, that is specific to the health care domain is developed. In the second level, health care and application-specific concepts such as blood pressure, lab results and so on are modelled as archetypes. Archetypes have constraint rules which specialise the generic data structures that can be implemented using the reference model. An archetype definition basically consists of three parts: Descriptive data, constraint rules and ontological definitions.

Integrating the Healthcare Enterprise (IHE) Cross-Enterprise Document Sharing (XDS) integration profile

Integrating the Healthcare Enterprise (IHE), a not for profit initiative, was founded in 1998 in the USA by the Radiological Society of North America (RSNA) (Siegel and Channin, 2001) and the Healthcare Information and Management Systems Society (HIMSS). The goal of the initiative is to stimulate integration of health care information resources by selecting and recommending appropriate standards for specific use cases. IHE develops restrictions, which are application profiles for standards that allow for a simplified system integration.

The basic idea of IHE XDS is to store healthcare documents in an ebXML registry or repository to facilitate sharing of the documents. IHE XDS is not concerned with document content; it only specifies metadata to facilitate the locating of documents.

In the IHE XDS integration profile, a group of health care enterprises that agree to work together for clinical document sharing is called the Clinical Affinity Domain. Such institutions agree on a common set of policies such as how the patients are identified, how the access is controlled and define a common set of coding terms to represent the metadata of the documents.

IHE XDS handles healthcare documents in a content neutral way. A document may include any type of information in any standard format such as simple text, formatted

text (e.g., HL7 CDA Release One), images (e.g. DICOM) or structured and vocabulary-coded clinical information.

Integrating the Healthcare Enterprise (IHE) Retrieve Information for Display (RID)

IHE defined RID as a web service by providing its WSDL description with binding to HTTP GET method. The WSDL description of different information sources may vary depending on how they implement the RID features, but the generic template provided by IHE very much restricts the possible variations.

Digital Imaging and Communications in Medicine (DICOM) and Structured Reporting (SR)

Digital Imaging and Communications in Medicine (DICOM) (Mildenberger et al., 2002) is known as the default standard for medical image communication. The standard defines data structures and services for the vendor-independent exchange of medical images and related information. In the year 2000, an extension to the DICOM standard that covers medical reports and other clinical data (Sluis et al., 2002) was officially released.

Structured Reporting (SR) is a general model for encoding medical reports in a structured manner in DICOMs tag-based format. It allows the existing DICOM infrastructure network services, like storage or query/retrieve, to be used to archive, communicate, encrypt and digitally sign structured reports with relatively small changes to existing systems.

The Medical Markup Language (MML)

The Medical Markup Language (MML) (Guo et al., 2004) was developed in the mid 1990s by the Electronic Health Record Research Group of the Japanese Ministry of Health and Welfare to provide a standardised way to exchange medical documents and other clinical data. MML uses the XML-based HL7 Clinical Document Architecture as a standardised container that carries MML information.

The survey concluded that there was no clear standard that addressed all the needs required and noted also that there was still a lot of work being done by various standards body to produce interoperable EHR standards. There is a working group committee under the ISO TC 215.

2.3.5 ISO standards

The International Organisation for Standardisation (ISO) is a network of national standards institutes of 157 countries, on the basis of one member per country, with a central secretariat in Geneva, Switzerland. ISO standards are technical agreements which provide the framework for compatible technology worldwide (Goetsch and Davis, 1998).

They provide a reference framework or a common technological language between suppliers and their customers which facilitates trade and the transfer of technology. This is achieved through consensus agreements between national delegations representing all the economic stakeholders concerned like suppliers, users, government regulators and other interest groups, such as consumers.

ISO standards are developed by technical committees comprising experts from the industrial, technical and business sectors which have asked for the standards and which subsequently put them to use. These experts may be joined by others with relevant knowledge such as representatives of government agencies, testing laboratories, consumer associations, environmentalists, academic circles and so on.

The experts participate as national delegations, chosen by the ISO national member institute for the country concerned. These delegations are required to represent not just the views of the organisations in which their participating experts work, but also of other stakeholders. According to ISO rules, the member institute is expected to take account of the views of the range of parties interested in the standard under development and to present a consolidated, national consensus position to the technical committee.

2.3.6 ISO TC215 Health Informatics Committee

According to (Kalra, 2004), ISO develops health informatics standards through technical committee ISO/TC 215 Health Informatics. The ISO TC215 Health Informatics

Committee is responsible within ISO for standardisation in the field of information for health, health information and communications technology to achieve compatibility and interoperability between independent systems. The aim is also to ensure compatibility of data for comparative statistical purposes (e.g. classifications) and reduce duplication of effort and redundancy.

TC215s standards development work is carried out through the following Working Groups

WG 1 Data Structure This group is tasked with developing standards that establish the structure of health information in order to facilitate the sharing of information and data among enterprises, organisations and information systems. These standards establish the definitions, context, organisation (framework and models), relationship and template requirements for health information and associated data sets.

WG 2 Data Interchange This working group is responsible for standardising the means of messaging and communication in health informatics such how the electronic exchange of information between individual systems (clinical and administrative) and organisations (clinical and administrative) is facilitated. This group usually meets as two separate streams: one focused on methodology, the other on architecture. It is the working group that has carriage of ISO adopting HL7, IHE and DICOM standards.

WG 3 Health Concept Representation (terminology) The scope of this working group is to develop standards for representation of health concepts and data. These standards include formal models of representation and description of health concepts; principles of their organisation within terminologies and related systems (including controlled clinical terminologies and classifications); and issues concerning context of their use in EHRs. Efforts are focused on selection and integration of terminological content within health Informatics applications.

WG 4 Security This working group defines guidelines for security management in health care and defines standards for technical and management measures to:

1. Protect and enhance the confidentiality, availability and integrity of health information.
2. Prevent health information systems from adversely affecting patient safety.
3. Ensure the accountability of users of health information systems.

WG 5 Health Cards

From conception, this working group was tasked with producing standards for health care usage of machine readable cards compliant with the physical characteristics defined in ISO/IEC 7810. The working group places special emphasis on technology independent data structures that can lead to interoperability, compatibility and communication of data cards used to identify patients and health care providers as individuals so as to support systems access and health record linkage.

The group is also focused on patient data cards intended to convey health care data of medical importance that may not be immediately available or usable by other means. The group sought to co-operate with other working groups in relation to data structures and not produce data models where such data models already exist.

Following broad consideration and discussion of recommendations from the working group itself, it was decided that the WG 5 health cards be retired so that it would no longer continue operating as a separate working group within the TC 215 structure, but that their work will be spread among other working groups as required. It was also noted that ongoing work will no longer be limited to magnetic stripe health cards and that smartcard, thumb drive and other media technologies need to be accommodated.

WG 6 Pharmacy and Medication Business

Scope of this group is to establish standards in the domain of pharmacy and medication including research, development, regulation, supply, use and monitoring in order to improve the efficiency and interoperability of medication information systems affecting patient safety.

This working group provides appropriate domain expertise to ensure that the business requirements for international standards in this area are identified and met either by cooperation with other groups (through adoption of their standards into ISO) or by development of new standards and technical reports within the working group.

The WG 6 also has a watching brief to monitor the need for health informatics standards and advise other TC215 WGs on requirements in the area of e-pharmacy and applications relating to medicines.

WG 7 Devices

This working group works on the standardisation of the application of IT to medical devices for plug-and-play interoperability at the point of care as well as facilitating the efficient exchange of devices in all health care environments.

WG 8 Business Requirements for Electronic Health Records

This working group is tasked with the standardisation of the identification of business requirements for all health informatics aspects applicable to health records for all.

ISO/TC 215 WG 9 - JWG for SDO Harmonisation (TC 215/HL7/TC 251)

ISO TC215 also provides secretariat services for the Joint Initiative Council (JIC) that manages collaboration between ISO TC215, CEN TC251 and HL7 under the Joint Initiative on SDO Global Health Informatics Standardisation. In addition, the Joint Working Group (JWG) that supports the JIC is constituted as ISO TC251 WG9, with Australia playing a major role by providing the secretariat.

The scope of the group JWG involves planning, process determination and coordinating group that makes recommendations to the Joint Initiative Council (JIC) on resolving gaps, overlaps or issues of counterproductive standardisation in health Informatics by:

1. Identifying, analysing, defining and documenting specific gaps, overlaps, issues and tasks to be addressed.
2. Using use cases addressing all parts of the standards life cycle.
3. Developing, testing and using effective decision processes for international standardisation needs.
4. Developing common processes for harmonisation in accordance with participating SDO processes.

The JWG is tasked with developing an integrated work program amongst the participating SDOs for approval by the JIC, including:

1. Collection and summarising of participating SDO work plans.
2. Building awareness of relevant standards activity.

3. Reviewing the work plans of participating SDOs gaps, overlaps and counterproductive standardisation.

4. Monitoring and providing feedback on the outcomes of the joint initiative

5. Encouraging stakeholder engagement and communicating output of the work programme.

2.4 IT infrastructure and Health Information Systems in Africa

2.4.1 Significance of networked application architectures

The advent of the internet has given rise and acceleration to the pace and development of application that can be shared among several people in different locations (Hegering et al., 1999) .

Traditional applications have been developed in a stand alone architectural style in which a single computer contains all the software components related to the graphical user interface (GUI), application service processing and persistent data resources. The flow of control in a stand-alone application resides solely on the computer where execution begins (Schmidt, 1994) .

In contrast, networked architectures divide the application system into services that can be shared and reused by multiple applications. To maximise the effectiveness and usefulness, services are distributed among multiple computing devices connected by a network. The networked architectural style partitions the interactive GUI, instruction processing and persistent data resources among a number of independent hosts in a network. At run time, the flow of control in a networked application resides on one or more of the hosts (Schmidt, 1994). All the system components communicate cooperatively, transferring data and execution control between them as needed.

Interoperability between separate components can be achieved as long as compatible communication protocols are used even if the underlying networks, operating systems, hardware and programming languages are heterogeneous (Schmidt and Vinoski, 1997).

The reasons for heterogeneous systems in distributed networks range from cost justifications, engineering availabilities, trade-off and use of legacy systems. This delegation of networked application service responsibilities across multiple hosts can yield benefits such as the sharing and rapid dissemination of information to more potential users.

Connectivity avoids the need for manual information transfer and duplicate entry. Furthermore, improved performance and scalability allows system configurations to be changed readily and robustly to align computing resources with current and forecast system demand. Further, networked applications have reduced costs because they allow users and applications to share expensive peripherals and software such as sophisticated database management systems (Mori et al., 1986). To add to these benefits, there is some security in networked architectures (Greenfield and Short, 2003). If one host is attacked by malicious programs, the other hosts have the chance to keep operating and not all information is lost .

For all these benefits to be fully realised, the underlying network infrastructure that supports this set up needs to exit. It must be reliable and robust with all the necessary redundancies to ensure that it is available most of the time.

2.4.2 The African Context

A service oriented architecture solutions involves the design and deployment of networked applications. The importance of using networked applications to deploy distributed applications across geographical locations is well known. Truly functional networked applications require a robust network infrastructure if it is to function well. Africas network infrastructure is fraught with reliability problems caused by a number of factors. It is therefore important that systems developers, building applications for deployment in Africa, construct applications that can function in an environment where the network infrastructure is unreliable.

As stated in the introductory chapter, Africa has been seeing a steady increase in ICT systems deployed in various institutions. This is evidenced by the growth of the sector. In his report, (Jensen, 2002) shows how the environment for networked readiness, with help from wireless technology from cell phone companies, has improved relatively

rapidly in most urban areas in Africa. The deployment of applications to take advantage of this improvement has also increased.

Despite this improvement in the landscape of ICT in Africa and the proliferation of deployed IT applications, the infrastructure connectivity on which these applications run, is still segmented. (Mutula, 2008) in his work confirms that most applications in Africa are deployed in areas that are not easily accessible or connected using ICT technologies.

Most parts of Africa lack the availability of a reliable connectivity infrastructure (Kenny, 2000). Because of this, collection and consolidation of data from various places will likely remain a challenge for a long time. It is imperative that the architects of IT systems take this into consideration and design application architectures that can function well in disconnected set ups, an environment where there is an unreliable network connection between the systems or nodes in a network topology.

2.4.3 Challenges of Networked Architectures in Africa

Networked architectures pose a huge challenge when it comes to deploying them in Africa. As concluded in section 2.4.1 , a networked architecture needs a reliable functional network to operate with minimal downtime. Downtime is the time in which a resource is not available to computer nodes participating in a networked environment. African has a number of challenges that affect the network. These range from poor connectivity, lack of electricity, regulatory framework and lack of guaranteed quality of service.

Lack of electricity makes it impossible to even power ICT devices like computers, network switches and transmitters that are crucial in a networked environment. In some areas, the electricity is so unreliable (Hodges et al., 2007) that constant power black out leads to some damage of the equipment used in the network architecture and replacement can be costly for the majority of projects that have constrained budgets.

Connectivity is also a major issue. Majority of the place have very poor connectivity infrastructure running on that is not properly maintained (Nixon, 2007).

Africas regulatory framework has also made it impossible to introduce competition that could help with quality of connectivity. Most telecommunication companies in Africa are

centred around state owned companies and few private ones that do not really compete in the provision of quality of service (Wallsten, 2001).

Africa also suffers a shortage of skilled personnel in ICT to build, deploy and later manage applications of this scope. Generally, there is a global shortage of skilled IT professionals. Africa finds it hard to keep its home grown talent or even attract the skilled people from the developed economies. A study done by Mutula (Mutula and P, 2007) found that there is a serious skills gap for certified ICT specialists to help develop the sophisticated applications necessary to power the digital economy in developing countries.

With all the above issues it is really a challenge to deploy networked applications that can function within the acceptable metric measurements.

Designers of any service oriented architectures have to take the above issues into consideration if they are to come up with designs that are practical in an African context.

2.5 Summary View of Literature Review Issues

From the literature above, it can be seen that there are a number of challenges one has to navigate in order to develop a solution that is sustainable in Africa.

The landscape shows three systems that one has to be aware of when working on a solution that addresses the architecture needs of the African environment in addition to skills gap issues.

The majority of deployed systems in Africa are designed on an ad hoc basis, primarily to solve the pressing needs and so tend not to have any standards at all.

The other group of applications that a developer of a new system will have to be aware of are those funded by the donor community and run by donor organisations across the continent. These systems tend to be transplanted from the donor countries and will have a number of heterogeneous standards, a reflection of the countries where they are imported from.

The work of the ISO TC 215 will have a major impact on the applications deployed across the continent once completed. Some African countries sit on these committees

and it is important that the developers of architectures that will be deployed in Africa keep an eye on the work being done by the working groups of the TC 215.

Another important aspect particular to an African environment is the connectivity issue. One has to be aware of the fact that the connections between the different systems is unreliable and subject to high segmentation levels. Architectures deployed need to be able to operate in a disconnected environment.

Technologies, particularly programming languages used to develop systems, needs to be based on what the teaching industry, colleges and universities is providing to students in Africa.

Figure 2.2 depicts the African landscape issues or challenges revealed by the literature survey.

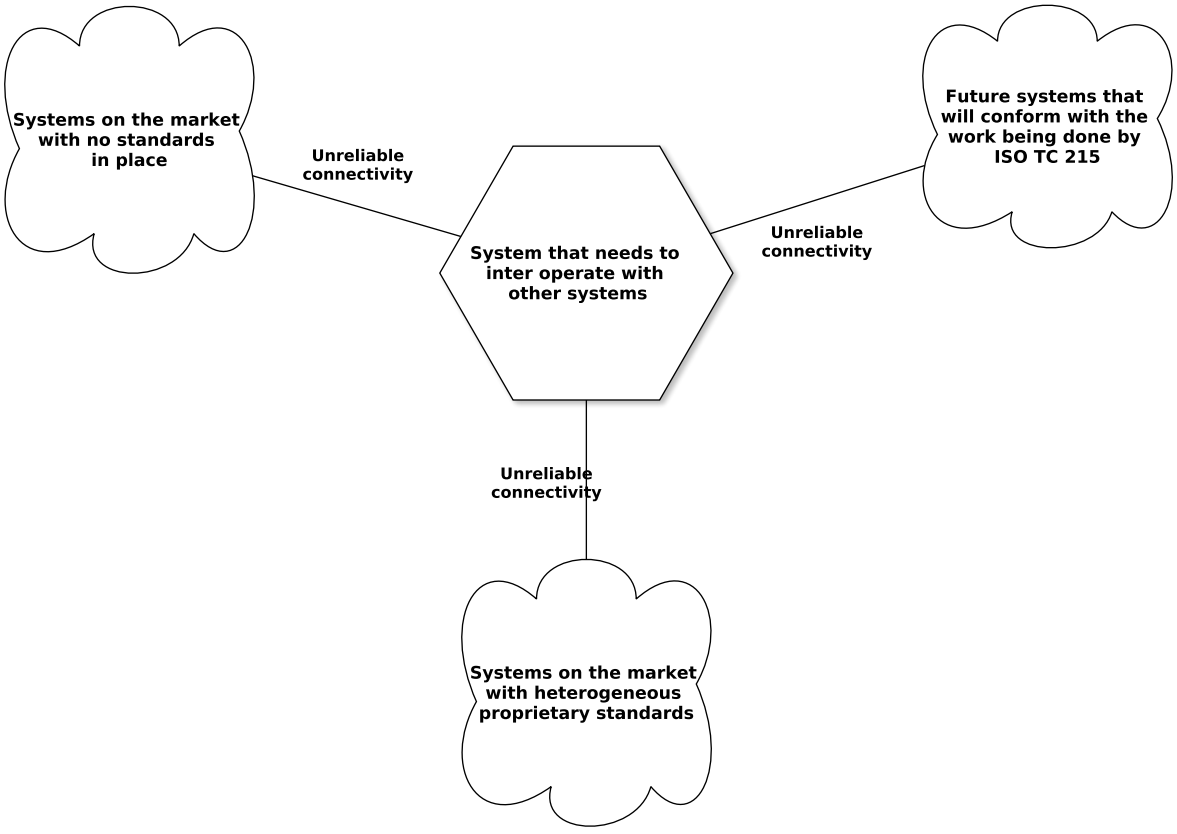

FIGURE 2.2: The landscape that any new system will have to deal with in an African environment to be able to interoperate with others

Chapter 3

Research Approach

3.1 Introduction

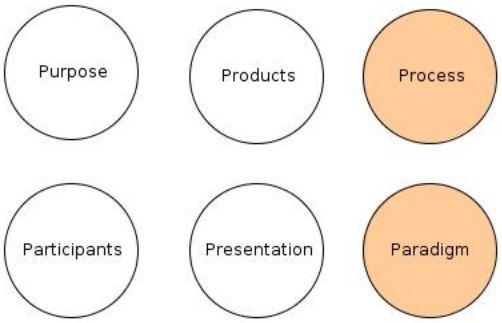

FIGURE 3.1: Aspects needed to be supported in Every Research

In this work we agree and support the view and approach held by (Oates, 2005) that in every research, the aspect of purpose, products, presentation, participants, process and paradigm needs to be supported.

The first four aspects are very clear and common among various research undertakings in various domains. The of purpose of this work has been articulated in chapter 1 in the problem section together with the product that will result at the end of the process. The participants are the people whose systems will be evaluated and people who will test the developed artefact in their environment. The presentation of the work will be done through the publication of this thesis.

The last two aspects, namely process and paradigm is where most of research work tends to substantially differ and we shall cover the two in more details to show how this work fits in.

This work's problem is premised in the field of information systems and the solution involves the construction of a software artefact to address the problems addressed in the previous chapters, hence making this an engineering problem. As rightly stated by (Hevner et al., 2004) in their paper, information systems and the organisations they support are composed of people, structures, technologies and work systems. These structures are deliberately created to meet the goals of an organisation.

Therefore, the research model for this work is structured to address a numbers of challenges faced by developers in building software artefacts and deploying them in an environment to see if indeed progress has been achieved by the newly developed artefact.

Additionally, the model has a second dimension that allows the development of the understanding of how and why the artefact works. This dimension allows us to tie together the natural laws governing software artefacts with the laws governing the environment in which the artefact is deployed.

There is also a third dimension that requires the studying of the impact of the artefact in the deployed environment. This social dimension is beyond the scope of this work.

This work is focused on the design activities of building the IS infrastructure within an organisation to solve the identified problems. This work is in a problem solving or engineering paradigm that has to produce an artefact. This process involves a sequence of expert activities that produces an innovative product. The evaluation of the artefact then provides feedback information and a better understanding of the problem in order to improve both the quality of the product and the design process.

3.2 Research Paradigms

Research paradigm is the fundamental set of assumptions adopted by a professional community that allows its members to share similar perceptions and engage in commonly shared practices. Typically, a paradigm consists of assumptions about knowledge and how to acquire it, and about the physical and social world.

The view taken in this work comes from work done and presented by (Korpela, 2013) on science categorisation. In the presentation, (Korpela, 2013) categorised sciences into three areas namely natural science, constructive science and human and social science. While philosophy is fundamental to the human and social sciences, it is not central in natural and constructive sciences.

Constructive science is the basis for this work and in this paradigm what is fundamental is the practice and largely what works is deemed to be true. The practice in this field gets its justification from the social world where long term effects are studied.

Before we delve into the details of the type of research done in this work, we shall explain our view of research and set the stage for the rest of the chapter.

3.3 Research Process and Model

A research process lays down the sequence of activities to be undertaken and is guided by a research model. It is the systematic execution of this process by following the model that makes research work accepted as rigorous. The diagram in figure 3.2 shows the model of this research. This figure is a revised presentation of the model proposed by (March and Smith, 1995) in their paper. The new model presented here shows two key research dimension of constructive science, which houses disciplines like design, engineering and natural science.

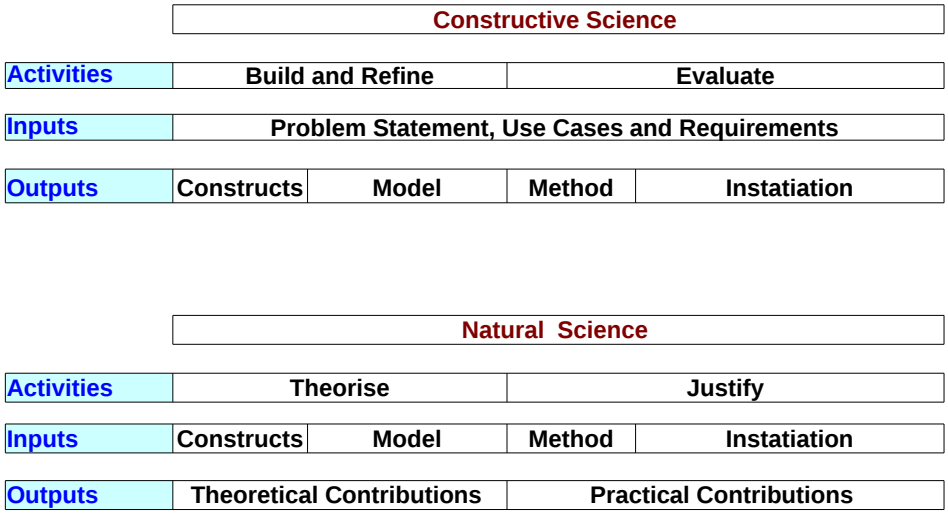

FIGURE 3.2: Research Process Model

3.3.1 Constructive Science Dimension

Information technology ecosystem is largely concerned with the development, implementation, operation and maintenance of systems. The design activities are largely driven by the development and maintenance of systems.

Design science, a subset of constructive science, is mainly concerned with attempts to create things that help with the attainment of goals for organisations. Design science is a technology oriented process whose products are assessed against a criteria of value or utility, answering questions such as does it work or is it an improvement.

The main activities during this process are build and evaluate, with the evaluation providing feedback for further refinement of the product.

Build and Evaluate

Building and evaluation are the core activities that drive the design science paradigm.

Building is the process of constructing an artefact meant to address a specific need identified by either the problem statement, problem definition, use cases or requirements. The build activity demonstrates feasibility that a proposed artefact, which becomes an object of study, can be constructed. The build process must adhere to the rules and laws in the construction of the software artefacts. Research contribution lies in the novelty of the artefact built. Once constructed, the artefact must be evaluated.

Evaluation is the process of determining how well the constructed artefact produced by the build process performs so that progress can be determined. The key question is how well the constructed artefact works. Evaluation requires the development of metrics against which the artefact can be evaluated. If the artefact meets the criteria set, it is then used to replace the existing one in place and progress is deemed to have been made because a more effective and better performing artefact is now in place.

Outputs

There are four outputs of the constructive science model as shown in the Figure 3.2. These are constructs, models , methods and instatiation.

In this work, constructs refers to the ubiquitous language and the UML use case model used in the construction of the solution. The use case model captures the activities of all the key players and the language used in the health care industry. It is from the use case model where the entities and relationships are mined to create a class diagram model.

Models refer to the class Diagram model and entity relation diagrams which depicts the solution in an object oriented and database world. This is the representation of what property and behaviour the models will exhibit and support.

Methods are services, algorithms and data structures that make the solutions work. These involve the combinations of different entities from the models to accomplish a particular task.

Instatiation is the actual product that is deployed in the environment to solve the identified problem. This could be an application or just an API providing a service to applications.

3.3.2 Natural Science Dimension

Information technology impact potential, be it negative or positive, has drawn scientists to take notice of the pervasiveness of what is described as an IT artificial phenomena described by (Drucker et al., 1988). The produced phenomena can be explained by scientific theories and so lead to the understanding of the nature of software artefact as a knowledge producing entity that corresponds to natural science based on natural phenomena (Hempel, 1966).

In this work, natural science, which is descriptive in explanatory and intent, is applied on the artefacts produced by the constructive science process which gives rise to the artificial phenomena that is a target of natural science study.

Natural science, according to (Kaplan, 1964), is viewed as a field comprising two activities called discovery and justification on a broad basis.

Given an artefact that has been successfully built and evaluated, it is vital to determine why and how the activity worked or did not work in the environment it was deployed in. Natural science methods are applied to the created artefact to determine the reason the artefact either worked or did not work. The main activity is to theorise and then justify those theories about what was produced in the design process.

Theorise and Justify

The discovery process is applied on the artefact to either generate or propose scientific claims called theories in order to try and understand the functioning of the artefact. To explain the unique characteristics of the artefact and its successful deployment in the environment as measured by the evaluation process, there is need to understand the unique laws governing both the artefact and the environment in which it operates. This is the dimension that theories provides.

Testing these theories for validity is what is called justification. This process constitute the gathering of the evidence in the form of either mathematical concepts or data gathering mechanism to test the theory. Key activity is to perform empirical or theoretical mathematical proofs to test theories posed.

Outputs

The output of this process is a deep and principled explanation of the phenomena created by the constructive process. The explanation makes the claims consistent with the observed facts. This, in the end, provides a tool that can be used to predict future behaviour or manifestation. Outputs are classified as theoretical and practical contribution of the work.

Chapter 4

Service Oriented Architectural Model for African Contexts

4.1 Introduction

From the literature review in chapter 2, development and deployment of a Service Oriented Architecture (SOA) in Africa will have to address a number of key challenges.

The African context terrain, where the application will be used, is fraught with many difficulties. Storage of information in a manner that makes it easy to share, without causing mutation conflicts, is an aspect that needs attention if a system is to be deployed in an environment where is useful to users. Therefore, any SOA platform needs provide a service to its users.

Undoubtedly, an application designed as a service will also have to deal with the communication style that requires these services to exchange messages. For exchange of messages to work, messages need to find their intended destination.

Even when messages reach their destination, there is still the task of making the messages meaningful to the parties that are involved in the exchange. The extraction of the meaning of messages will have to happen through a process called mediation.

4.2 Data Storage Challenges

Data is everything and the key reason we build IT systems is to help with the clinical processes using the data that we have. Most applications deployed in the health care industry are for the sole purpose of collecting data, securely storing it, sharing the data and using it to make key decisions and plans. Because of this, data is placed at the centre of any application design.

Therefore, data is the common denominator in the ecosystem that promotes data sharing. To create a data sharing environment, there has to be a common mechanism to access the data that is shared without compromising the quality and integrity. You cannot talk about efficient data sharing without an efficient network infrastructure.

There are two situations to consider when dealing with common data sharing among the users. The networked environment where good quality connection is guaranteed and the disconnected environment in which connectivity is either poor or not available.

4.2.1 Networked Environment

In a networked environment, multiple users accessing and manipulating data at the same time is a core characteristic. In this type of set up, the data layer of any built application is bound to experience user requests attempting to edit the same data at the same time, especially when there is a single data source associated with the several access requests for manipulation like read and write from users. This competition for data is known as a concurrency condition and can result in a lack of integrity in the stored data or worse still, a loss of data.

The most common default storage engine for data in Africa that we have seen from most public software projects is a relational database. In relational databases, prevention of data corruption due to concurrency issues is done by locking the records being worked on. Optimistic and pessimistic locking are two approaches used (Guo et al., 2008). Optimistic locking is preferred in cases where concurrency conditions are low. Pessimistic locking is used when the concurrency conditions are high.

In a pessimistic case, locking is done by preventing a user from getting a record to edit when it is being edited by another user. This type of locking requires that a user

remains connected to the database for the entire process. This is impractical in a disconnected environment. Furthermore, this type of locking could hold up other users trying to access the data. This, in turn, can degrade the availability and usability of the information the user is trying to access.

In optimistic locking, multiple users can get the same record for editing. Locking kicks in after one user tries to persist their changes on top of someone elses. Before or during an update, the application logic outside the data storage will check to see if the current record in the database has changed since the copy of the record being edited was retrieved. If it has changed, the data storage system will generate an error causing the update transaction to be rolled back. On the other hand, if no changes are detected, the record is successfully persisted into the data storage engine. Again, this type of locking requires constant polling of the underlying data storage engine for any changes. The availability of a reliable connection is also paramount.

4.2.2 Disconnected Environment

As mentioned before, if you are depending on a relational database system to maintain the locking mechanism, you will need to stay connected to the data source throughout the process. This mechanism of having to stay connected throughout the transaction process may not be possible for all types of applications, particularly those deployed in African environments where you have issues described in section 2.4.3.

Implementing your own application locking mechanism for relational databases in a disconnected environment can be a challenging exercise. One may need to provide a mechanism for reversing wrong changes made by several users over time. This is because, in a disconnected environment, it is very likely and possible that someone may get a record and work on it for a very long extended period of time when they are off-line and others may repeatedly make several changes to that record.

Furthermore, in a disconnected environment, there is no upfront indication to the user that someone else may be editing the records or may have retrieved the record for editing. Depending on the circumstances of your data consolidation process and the number of users allowed to change records at the same time, the design of a solution to deal with these problems could be an intimidating and long process.

Any solutions designed to work in a disconnected environment must address these problems that will arise if the network is segmented. There will have to be an audit trail that can be reconciled manually when all the nodes have shared all their data changes.

4.3 Services Challenges

The health care environment has a lot of legacy systems and new systems that are incompatible with each other. Most solutions tend to re-use already existing legacy systems by delegating functionality to them. Legacy systems implement complex business logic for which a complete rewrite would be an unjustifiable cost.

Also, the current trend now in design of bigger applications is to divide big systems into into multiple components that run independently, sometimes even located on different machines.

What is common in both cases is that systems are being integrated together as separate components. The key challenge is to make sure that the integrated components do not impose needless restrictions on each other.

It is important to understand that for the system to function properly and allow its constituents parts to evolve, the components taking part in this marriage are expected to evolve independently. Without this, the evolution of a system will become prohibitive. The design of this ecosystem should make this process easy. It is very important to recognise concerns that can make evolution prohibitively hard and work to avoid them.

One measure used to decide whether a particular design strategy fosters independent evolution is what is known as coupling. Coupling is the measure of how many dependencies the connected systems have on each other and how tightly connected the parts of a system are, including the assumptions they make about each other. There are two sides to coupling: Tight coupling and and loose coupling.

In tightly coupled systems, the degree of dependencies is high and the components involved in the system cannot evolve independently, making the systems impossible to adapt to the ever changing requirements of the end users. In loosely coupled systems, the dependence degree is lower and systems depend on interfaces which make the components evolve independently.

Interfaces give rise to the notion of services or contracts that abstract the implementation of services away from the actual service that is exposed to consumers as a dependency.

4.3.1 Service as an Application Programming Interface (API)

As stated in the literature survey, the heterogeneous environment in which systems operate gives rise to a situation where clients produce different data formats and request different data formats too. It is important that we set a standard on how data can be accessed and advertised to the potential users. This standard is what is called a service and needs to act as a basis or contract with whoever collaborates with an application that is advertising a service.

A service will be a software function that carries out a business task and provides access to our data. The key challenge is dealing with how services can be used to share logical functions across different users in order to enable client applications, which run on disparate computing platforms, collaborate.

From the literature, it can be seen that a standard based on web services provides solutions in this space. Web services provide the means to integrate disparate systems and expose reusable business functions over HTTP. Web services either leverage HTTP as a simple transport over which data is carried like SOAP/WSDL services or use it as a complete application protocol that defines the semantics for service behaviour such as REST based services.

Because these technologies use open standards, they provide a ubiquitous and inter operable solution across different computing platforms independent of the underlying execution technologies.

Despite the fact that all web services minimally use HTTP and leverage data-interchange standards, there is still the challenge of choosing which type to use. Web services use HTTP in two distinct ways. Some use it as an application protocol to define standard service behaviours while others simply use HTTP as a transport mechanism to convey data.

Additionally, there are other challenges like service creation, deciding on a common API to use, determining the binding style mechanism to use (RPC-encoded,RPC-Literal,Document-Encoded,Document-Literal and Document-Literal-Warped), determining the clients registry services and many other security and upgrade issues.

4.3.2 Communication Style

Having services that are well designed and complying to standards is one thing and guaranteeing that the messages sent out by those services reach their intended destination and receive acknowledgements is another. With the poor infrastructure of the network in Africa, guaranteed message delivery needs to be provided. In such an environment, a typical remote procedure type of communication that requires the network to be constantly available cannot be an answer to the architecture that we are trying to build for deployment in a disconnected set up.

The Communication style that we shall need in this case will be messaging. Messaging is a technology that enables the reliable delivery of needed information from system to system. This is the type of infrastructure that will adapt to the network environment found in Africa.

4.3.3 Message Routing

Any system with numerous players and destinations will have several paths which messages will pass through to reach the final destination. The route a message must follow may be so complex that the original sender does not know what path the message will use to the final target. Typically the sender sends the message to a message middleware which will determine how to navigate the paths.

This is the crucial question that must be answered if one is to deploy a functional message based infrastructure because messaging is inherently stateless, a quality that makes it scale. Therefore, for this to work, there are certain things that the application must provide.

First, message routing should be used to move data in different ways based on conditions. A routing mechanism should be able to forward messages to multiple end points or determine which end point from among many is to receive a message.

Secondly, message filter should be used to complement a routing mechanism in determining whether an incoming message is to be forwarded to a target or not.

To reduce load on the clients, routing should be centralised to make management of routing information happen in one place so that the routing decision logic is encapsulated in a single component. The routing part does not need to know anything about what follows downstream or upstream; the only information they require is which channel they should send the message to.

The challenge here is to remove as much infrastructure related knowledge and decision responsibility as possible from the individual system. The central system's role is to choose a next target channel for a message.

4.4 Mediation Layer Challenges

Systems that need to be interoperate by means of a messaging system or any other mechanism rarely agree on a common data format. This is one of the core issues at the heart of this work. For example, one system may persist data in a relational data model while another application may use flat files or XML documents as persistance storage.

Therefore, the integrating solutions will have to accommodate and resolve the differences between the varying systems. This is the role played by the mediation layer.

Key operations taking place in mediation is translation and enhancement. Translations offers a general solution to such differences in data formats. What is involved here is changing data from one format to another. Enhancement deals with adding extra information to the incoming data to meet the processing needs of the intended target.

4.5 The SOA Architecture for African Context

An application needs to address three key things to make it work in an African context while delivering the same services that traditional SOA applications are supposed to deliver. The first two involves the way the messages move into the system an out of the system to the rest of the world. The final piece, which takes place in the mediation layer, involves the mediation and routing of messages to appropriate channels.

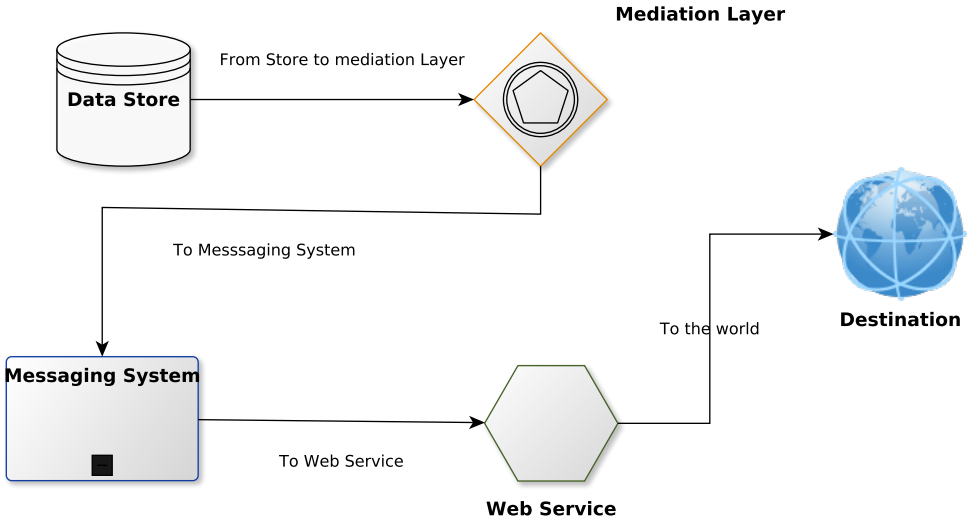

FIGURE 4.1: The path taken by the flow of the messages out of the system

Figure 4.1 shows the flow of messages out of the system to the consumers. Since the decision nodes like mediation layer, messaging system and the Web service have bidirectional channels, the consumer will also have the similar set up except that the flow of messages is reversed.

The mediation layer can be modelled as a re-usable component which not only keeps track of the messages being moved, but also acts as a reference point. Messages can be pushed to this component and pulled from it by clients as consumers. All conflicts are managed by clients when they push messages and pull messages.

The mediation layer has two channels for incoming and outgoing messages. Depending on the configuration, messages pass through to other connected clients or message store.

4.5.1 Interoperability solutions at the Data Storage Level

The storage discussed in this section is global data that is shared by all systems involved in a cluster. This is storage for replicated or shared data such as patient records, system users, appointments and so on. Storage used by the system for routing, logging and messaging middleware is never replicated.

First, the data storage that all the clients use has to function in a disconnected architecture. Distributed client nodes sharing a common data store need to operate despite the central data store servers inaccessibility. The clients nodes need to operate by emulating data provided by the central data store services locally. The nodes should be independent and should have independent copies of the central data store. At some point in time, when all the nodes in the network detect the availability of the network, they need to do a bidirectional share of the changed copies on each others nodes until all the nodes involved in the network topology have the same data.

In this architecture set up, the database is in a peer-to-peer set up and not the master-slave mode. In a peer-to-peer set-up, all the nodes equally share the data and the central server involved in the network. Data changes can be generated on any node and those changes need to be copied or replicated to the rest of the nodes participating in the network topology.

In a master-slave set up, the master generates most of the changes and just copies the changes to the slave nodes. The slave nodes, in most cases, only provide read services to the clients. This is the type of service largely used in backup systems.

This concept of disconnected architectures is not new, particularly in the mobile computing environment as shown in (DeRenzi et al., 2007). In their paper, they described the challenges faced by disconnected environment, which are very similar to problems faced in the environment of most African countries where the network is problematic or non-existent. When you have a scenario where network segmentation can exist for a long time, there is need to find a strategy to manage data updates. One major point that stands out in such an environment is the management of conflict detection (Green et al., 2007).

Solutions for Disconnected Data Store

The solution to disconnected data store is to use a process known as bi-directional replication, in which a change of a record on node is copied to the other nodes. Most relational commercial databases on the market provide this facility using complex protocols to manage conflicts described above.

There are three major conflict sources in a disconnected environment namely record update, new record creation and record deletion. A record update conflict occurs when the same record is updated on two or more different nodes before those updates can be propagated to the other nodes.

New record creation leads to what is would called uniqueness conflict. A newly created record can be assigned a primary key value that is already assigned by another node in the topology.

Lastly, deletion at any one node can cause a conflict on the other nodes. A delete conflict occurs when a record is deleted from one node, but when it tries to propagate the delete it cannot find the record in the remote nodes. This is either because the record has been deleted already or because values in that record have been updated.

Ideally, applications meant for disconnected architectures should be designed to avoid or minimise conflicts. Unique keys for the records should be global and if possible nodes should be allowed to change their own local data that they have originated.

However, this strategy is not enough in most environments where there is no such thing as local data and the only way to resolve conflicts is to look at the conflicting data and resolve it individually.

Recently, there has been an emergence of document oriented database solutions that manage this conflict better. Being document oriented makes it possible for these databases to use the versioning principles used in version control to manage conflicts and overwrites, which are a big problem in data shared space.

Management of conflicts are better managed by designing the application in such a way that there is minimum room for data conflict. If data conflict occurs, individuals can easily manage this. Great lesson can be learned on how conflict resolution is

managed by software programmers who work in a collaborated way using a version control system (Mukherjee et al., 2008).

The core mission of a version control system is to enable collaborative editing and sharing of source code or files by preventing users from accidentally corrupting each others work (Collins-Sussman et al., 2007). Unlike databases, which use a locking system to prevent conflict issues, a version control systems use a copy-modify-merge model as an alternative (Altmanninger et al., 2009).

In this model, each user's client creates a private working copy of a local reflection of data in the store. Users then work simultaneously and independently, modifying their private copies.

Finally, the private copies are merged together into a new, final version. The version control system often assists with the merging, but ultimately, a human being is responsible for making it happen correctly.

This is the same model that document databases use. Document oriented databases, unlike relational database, do not store records in tables but keep them as versioned documents. They are peer-based distributed database system with good support for replication. The replication allows for use in disconnected environments.

Once installed in different locations, users can access and update the shared data while disconnected and then bi-directionally replicate those changes later to the other nodes in the network.

Document based data storage solves the conflict problem by allowing for any number of conflicting documents to exist simultaneously in the store, with each database instance deciding which document is the winner and which are conflicts. Only the winning document can appear in views, while losing conflicts are still accessible and remain in the database until deleted or purged during some manual audit mechanism. Because conflict documents are still regular documents, they replicate just like regular documents.

This solution only applies if you want to interoperate or integrate your application at the data level where you have similar data store. If the integration and interoperability has to happen in an environment where you have traditional data stores that are different, then integration and interoperability will need to happen at the service layer where mediation of data formats take place.

4.5.2 Interoperability solutions at the Service Level

The scenario at the service level is that there exist deployed applications that function well, but need to share information with other systems. The first step to share information is to adhere to Distributed Systems standards of sharing data. As shown in the literature review, web services have proved to be a better standard for sharing information and building Distributed Systems.

The core function of a service layer is to expose the API of services it produces and wants to consume. In this work, the use of REST based web services for exposing services that are offered by applications is proposed.

From experience of the developed software artefacts, there are a number of things that need to be taken into consideration when designing an API that is REST based and usable.

Using REST Based Web Services

REST based web services tell us how the web achieves its great scale in terms of interoperability (Pautasso et al., 2008). A REST based Application Programming Interface (REST API) here will enable a client, either user-operated or automated, to access services or resources that model a systems data and functions. To build a REST based API, we need to take into account two key factors that make the world wide web work: statelessness and uniform interface.

Statelessness dictates that a system providing a service is not required to memorise the state of its client applications. As a result, each client must include all of the contextual information that it considers relevant in each interaction with a service system. A serving system asks clients to manage the complexity of communicating their application state so that a serving system can service a much larger number of clients. This trade-off is a key contributor to the scalability of the REST based architectural style and is something we have to bake in our SOA Library for African environment.

Uniform interface helps break down the communication barrier between serving systems and their clients. The interactions between the systems and clients depend on

the uniformity of their interfaces. If any of the systems or clients stray from the established standards, then the communication system breaks down.

All web services implementation in use today make use of HTTP protocol to transfer data. HTTP is a ubiquitous application-level protocol that defines operations for transferring data between clients and servers. HTTP's methods such as GET, POST, PUT, and DELETE are operations on resources on the serving systems.

The limited methods used by the HTTP protocol provides the constraint or limitation needed to create a uniform interface because there are few methods that are used to transfer data. This has made it easy to gain consensus among the players in the industry.

Please note that the benefit can only be achieved if HTTP is used as an application level protocol rather than a transport protocol. Web services based on SOAP uses HTTP as a protocol to transport messages. Such usage makes poor use of HTTP level infrastructure and so will not be used in this work.

In designing the API that communicates with other systems, we are going to follow the constraints identified from the literature about what constitute as a process for good API design (Alarcon et al., 2011)

1. Identification of resources.
2. Manipulation of resources through representations.
3. Self-descriptive messages.
4. Hypermedia As The Engine Of Application State (HATEOAS).

Resource Identification

Each distinct system concept, for example person entity, is known as a resource and should be addressed by a unique identifier such as a Universal Resource Identifier (URI) in REST based architecture. The resource is anything we expose to the clients for them to interact with while progressing toward some value goal, for example making an appointment. Almost anything can be modelled as a resource and then made available for manipulation over the network.

There are a number of ways we need to identify resources in existing systems or in new systems that we are building. The first place we identify resources is the domain model where the systems is based on and houses the knowledge of an application. Resources are also identified from use cases and problem descriptions. After the resources are identified, further analysis is required to identify nouns on which operations such as create, read update and delete can be used and these operations are mapped to the constrained methods of HTTP namely GET, PUT, UPDATE and DELETE.

Consider a web service for managing a patient appointment. Clients can create a new patient appointment, update an existing appointment, view an appointment or delete an appointment. In this example, appointment is an entity in the application domain. The actions a client can perform on this entity include create a new appointment, update an existing appointment, view an appointment and delete an appointment.

Next we can use HTTPs uniform interface to operate on these appointments as follows:

- Method GET to get a representation of each appointment.
- Method PUT to update an appointment.
- Method DELETE to delete appointment.
- Method POST to create a new appointment.

In complex scenario where the action needed goes beyond the simple create, read, update and delete, we can model processes as resources with input parameters and potential outcomes. The client system can provide values to the input processing function, which is acting as a resource and a client can get a client through representation (see next section). The same can be applied to decision systems, with the decision states modelled as resources.

Resource Representation and Manipulation

Clients manipulate representations of resources. The same exact resource can be represented to different clients in different ways. For example, a document might be represented as Hyper Text Markup Language (HTML) to a web browser, and as a

Media Type	Format	Reference
application/xml	Generic XML format	RFC 3023
application/json	Generic JSON format	RFC 4627
application/pdf	PDF	RFC 3778
text/html	Various versions of HTML	RFC 2854
text/csv	Comma-separated values, a generic format	RFC 4180

TABLE 4.1: Table showing extract from the IANA standard message formats

JavaScript Object Notation (JSON) to an automated program. The key idea here is that the representation is a way to interact with the resource but it is not the resource itself. This conceptual distinction allows the resource to be represented in different ways and formats without ever changing its identifier.

Most commonly used text formats in REST based system besides HTML and JSON is XML. XML, like HTML, organises a documents information by nesting angle bracketed tag pairs. Unlike HTML, well formed XML must have tag pairs that match perfectly, with a close tag to every open tag. JSON uses curly brackets to hierarchically structure a documents information.

Since there can be a number of ways in which representations can be represented, it follows that no single format may be right for all kinds of resource representations. A criteria to has to be found on how to choose a representation format and the media type to be used.

The first key criteria is to keep the media type as flexible as possible so that the unknown number of clients that consume a service can be served. The second key criteria is to determine whether the format chosen meets a standard format (Narten, 2008). This information can be obtained from the Internet Assigned Numbers Authority (IANA).

When selecting a format and a media type for representations, first check what clients you will be serving and see if the representation is available as a standard with IANA, whose registry lists media types by primary types such as text, application and sub types such as plain text, HTML, and XML.

4.5.3 Self-Documenting or Descriptive Messages

Because of the disconnected nature of the architecture, resources are required to have all the needed information to enable the client accomplish all the required tasks.

The messages transmitted do not need to have all the information required to do the task. Instead they need to have metadata that provides navigational information of where the needed resources can be obtained. The metadata convey additional details regarding the resource state, the representation format with the size and the message itself.

Key to navigational and accessibility information is the notion of Universal Resource Identifiers (URI) of resources that work across the network. The URIs has four parts. The first part is the scheme used to identify the protocol used to communicate the information. This could be HTTP, FTP or SMTP. The second part is the authority which identifies the party with jurisdiction over the namespace, usually the internet domain address. The third part is the path which identifies a single node within a larger and hierarchical resource model. The last part represents the parameters that can be added to provide the uniqueness of the scheme.

Domains and sub domains of organisations are used to logically group or partition resources for localisation, distribution or enforcement of other attributes like security and access policies. According to URI guidelines (Berners-Lee et al., 1998), forward slash separator (/) in the path portion of the URI should be used to indicate a hierarchical relationship between resources. Comma (,) and semicolon (;) should be used to indicate non hierarchical elements in the path portion of the URI. The hyphen (-) and underscore (_) characters should be used to improve the readability of names in long path segments. Ampersand (&) should be used to separate parameters in the parameter portion of the URI. File extensions (such as .php, .aspx, and .jsp) in URIs should be avoided.

Statelessness and HATEOAS

Our work uses REST as a web service communication infrastructure. Unlike SOAP based web services that use HTTP as a transport, REST embrace HTTP as an application protocol. One of the challenges faced when using HTTP as a transport protocol is the stateless nature of the protocol which makes the web scale by making the web forget its previous actions. Instead the infrastructure is connected together through a series of hyperlinks embedded within HTML documents. These links create threads between interrelated resources on the internet which can be used to navigate around it by clicking on links.

Besides links, another important feature of the web infrastructure is HTML forms which are used to collect data. An HTML form is a self-describing interaction between the players in the web infrastructure.

Each representation that the consumer receives from the producer represents the state of the users interaction within the application. For instance, when the user submits the form to receive another page, the user changes the state of the application from their point of view.

To change the state of the application, the user relies on forms and links found in the HTML. HTML is an hypermedia format, allowing link and form controls to let you flow through the application and thereby change the state of the application.

The architectural principle that describes linking and form submission is called HATEOAS. HATEOAS stands for Hypermedia As The Engine Of Application State.

Since a hypermedia system is characterised by the transfer of links in the resource representations exchanged by the participants in an application protocol, links advertise other resources participating in the application protocol. Links are enhanced with semantic markup to give domain meanings to the resources they identify.

The consumer is made to submit an initial request to some entry point of the service. The service will handle the request and responds with a resource representation populated with links. Then consumer will choose one of these links to transition to the next step in the interaction. Over the course of several such interactions, the consumer will progresses toward some specific goal.

So the current state of a resource is a combination of:

- The values of information items belonging to that resource.
- Links to related resources.
- Links that represent a transition to a possible future state of the current resource.
- The results of evaluating any rules that relate the resource to other local resources.

4.5.4 Messages

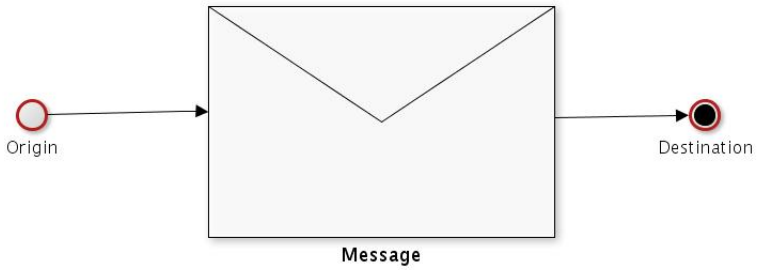

FIGURE 4.2: The Message

The web services described above will be receiving messages that will need to be passed over to downstream (destination) and upstream(source)services. Messages are the objects that carry information between two systems involved in the information exchange. They are typically independent units of information sent from one component of the software system to another. A piece of information is any data that can be represented in the system (objects, byte arrays, strings, and so forth) and passed along as part of a message.

Each message will contain all the information that is needed by the receiving component for performing a particular operation. Since messages are independent pieces of information, and messages can take on all sorts of formats, there is no way of imposing a standard on what these messages should contain. The agreed upon semantics of the messages being passed should be based on what the receiving consumer is expecting.

Messages are constructed at one end, usually the producer. They are then consumed and de-constructed at the other end, the consumer/subscriber. The publisher/producer

will create these message units and publish them to a channel. A subscriber/consumer connected to the same channel will then receive the message units. The domain objects are then resurrected from these message units and business processing is carried out.

4.5.5 Channels

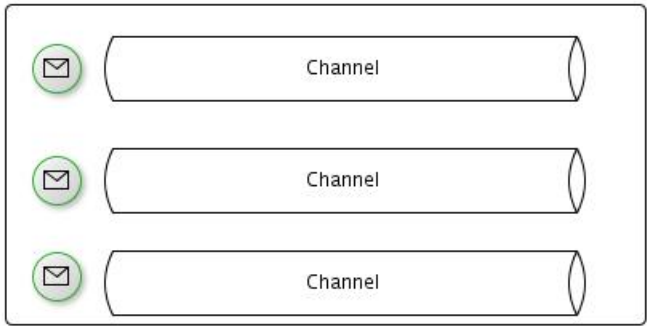

FIGURE 4.3: Message Channels

For messages to do something useful with the information they are carrying, they need to travel from one system to another. For this to happen, they need channels, which are well-defined conduits, to transport messages across the system.

Channels enable messaging consuming and producing systems to be decoupled from each other and only be concerned about what kinds of messages they can send and receive. Channels are points where messages are deposited or picked up.

While Message represents a container for information data, the channel represents the location where it is being sent. In other words, the message will end up at a prespecified address called channel before being used by someone else.

Channels are made up of two parts: the message channels and the message end points. The message channels will be used to connect the message end points.

4.5.6 End Points

End points are basically components that will consume messages from an input channel and deliver to an output channel by combining the business logic with integration specifications.

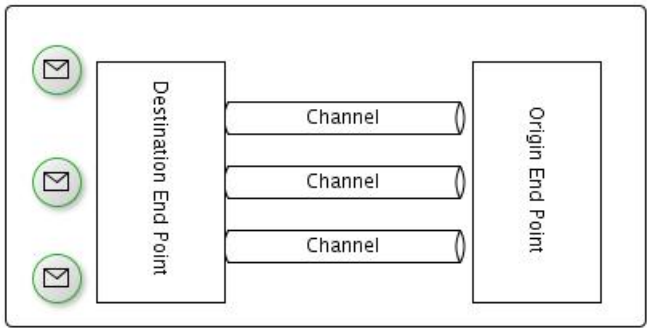

FIGURE 4.4: Message End Points with Channels attached

The message end points will be created in a declarative fashion so that the system behaviour can be altered based on the changing needs of the system situation.

Message end points, which are components that separate business logic from the messaging system are crucial in the integration space for hiding the messaging details. They are largely responsible for connecting application components to the messaging channels to send or receive messages.

The end point connected to a web service interface is made up of a number of components which have to deal with the incoming or outgoing messages.

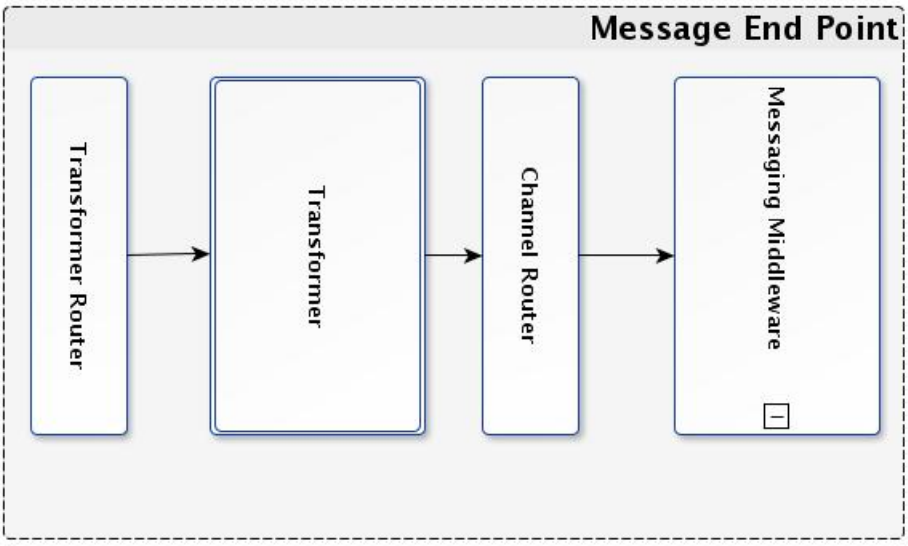

FIGURE 4.5: Message End Point Components

4.5.7 Transformation Router

The transformer router will direct messages to the appropriate transformation engine declared by adding messages to an external configuration file in the transformation engine to enable flexible configuration.

The transformer will have several engines to transform messages into what is desired by the destination channels. The channel router is there to select the appropriate channel which connects to the correct message destination.

To guarantee message delivery, provide the robustness and scalability needed by the system when moving messages from the routing channels, messages are delivered via the Message Oriented Middleware (MOM). The middleware is again declaratively configured or given the information about which channel to connect and deliver the messages to.

Once the incoming messages have been received at an end point, the next challenge is to route the message to an appropriate destination. Destination could include the source of the message as well because the routing engine would need to know where the message is coming from in case it needs to send back an acknowledgement.

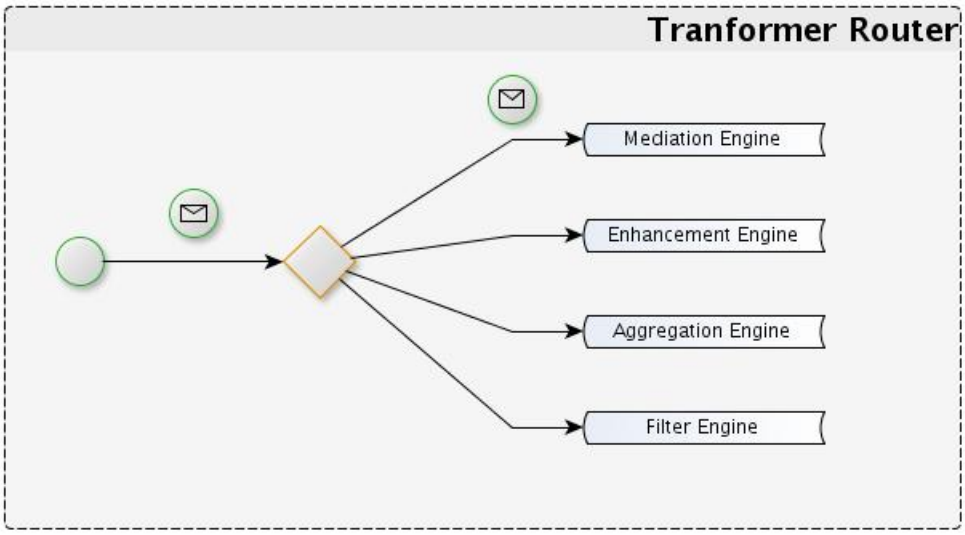

FIGURE 4.6: Transformer Router

One of the message flow requirements is to send the messages to one or more transformation engines based on certain criteria. The part played by the transformation router component is to distribute the messages to multiple destinations.

For example, if a message produced by a client has a patient record that is in some other format and is destined to an application that needs to consume HL7 message, the process to route this message will involve the transformer router. The router will first determine what type of message the client is producing and what type of format the destination consumes. Based on this information, the router in the above example will make the message go through the transformation router where the message can be transformed into the required HL7 message by the consumer.

The routing transformer will be querying its routes from a local store database. The use of the local store database allows the dynamic altering of several routes as needs arise. This also makes the application to continue to function without disruption as the routes are added.

Message Transformer

Message transformation is at the heart of integration. One crucial key challenge of integration is that the underlying data models of any two given applications are likely to be different or have incompatible data types.

The various properties can vary for a simple field such as an address. Some applications define the complete address as a single property and others break down the address into the street number, city, postal code and so on.

In addition, the exchange format of the record itself may be incompatible: system A might expect a normal XML file and system B might expect an HL7 XML-encoded record.

One client system might use a flat text as its payload to produce a message while a receiving system might be interested in plain XML or name-value pairs.

To help the producer and consumer communicate, transformers are used to transform one message format into the required format.

There are many transformation engines that can be added to the transformer. The above diagram shows three transformation engines, namely message translator, message converter and message enhancer.

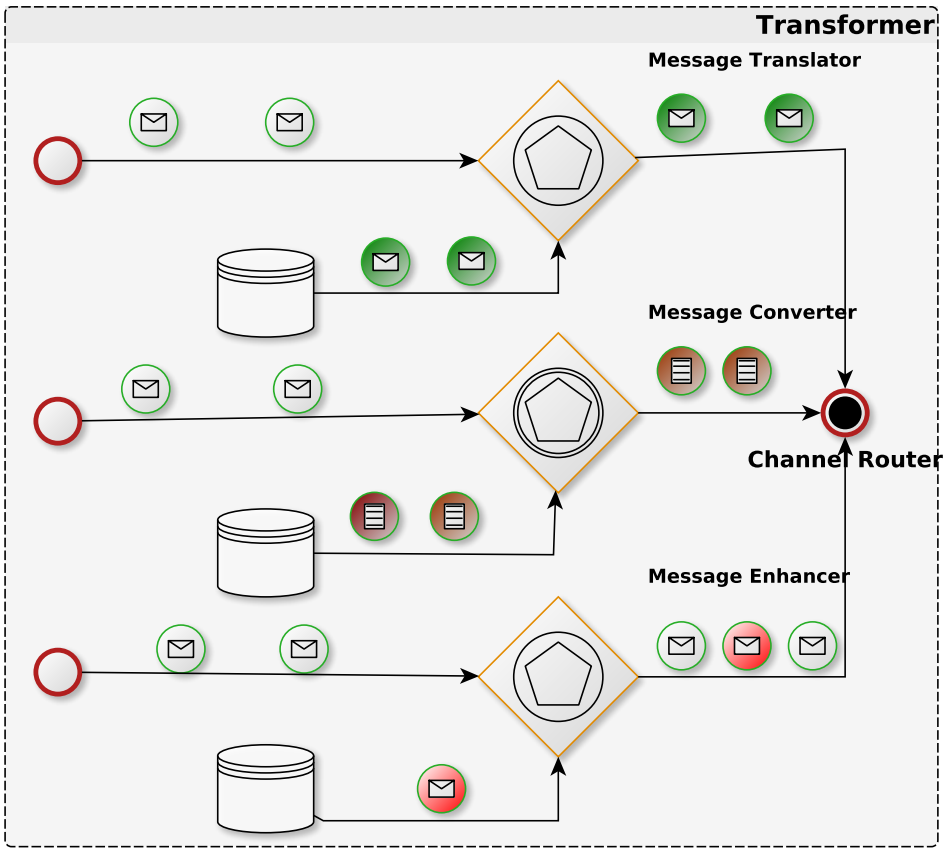

FIGURE 4.7: Message Transformation

Message Translator

The Message Translator is used for translating one message from one format to another. The translator keeps a record of both format used by the input and output. For example, the incoming message might store the milestone of a child growth as a key value string shown in the table 4.2.

	Variable Question	Variable Answer
Producer	Walks at a year old?	Yes
Consumer	Milestone passed	walks at a year old

TABLE 4.2: Key value pairs stored of data produced and consumed

The task of the translator engine will be to take the key pair values of the input from the consumer, translate it and pass it to the consumer in the required format for consumption. Undoubtedly, intimate knowledge of both consuming and producing application will be required. Again, these values are designed to be dynamically updated.

Message Converter

Not all systems understand the data they consume. Therefore, messages need to be converted before they can be consumed or used for a particular purpose.

For example, a producer may use a data format programmed in an object oriented programming language such as Java as its content or payload to produce a message while a consumer is interested in non-Java language format like plain XML or name-value pairs.

To help the producer and consumer communicate, message converters are used in the Integration end point to transform Java based format message to an XML based format needed by the consumer. The code snippet below shows the configuration of the conversion process.

```
1       <int-xml:marshalling-transformer
2             input-channel="object-java-in-channel"
3             output-channel="stdout" marshaller="marshaller"
4             result-type="XMLResult">
5       </int-xml:marshalling-transformer>
6       <bean id="marshaller" class="CustomTranslator" />
```

Listing 4.1: Code Snippet

In the code above, line 2 is the channel input that will receive a message in a Java format and line 3 is the output channel which is tied to a custom converter called the marshaller. Conversion is processed by the **CustomTranslator** shown in line 6 to output the XML format of the message received from the input.

Message Enhancer

There are times when the incoming message does not have the expected required information that the consumer needs. To solve this problem, a message enhancer engine is used to enrich the incoming message with additional information in order to send the updated message to the consumers.

The enhancer either uses information inside the incoming message header or body (payload) to determine what extra information is requited by the target consumer. This

information is used to retrieve the additional information required by the target consumer from an external source, which in this case is a local data store.

After the enhancer retrieves the required data from the resource, it attaches the data to the message. The original information from the incoming message is then carried over into the resulting message as shown in the above diagram for message enhancer.

4.5.8 Channel Router

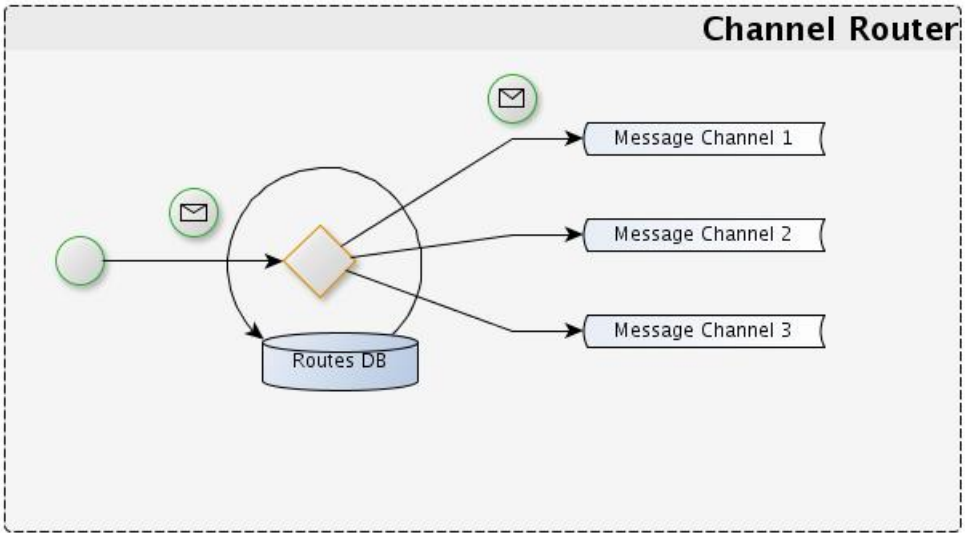

FIGURE 4.8: Channel Router with Routes stored in a Database

The channel router above works similar to the transformer router described earlier except that it forwards messages to a messaging middleware that has appropriate channels configured.

The channel router examines the message content and routes the message onto a different channel based on data contained in the message. The routing can be based on the destination headers of the incoming message. The channels are also configured at runtime to make the routing flexible to change when the application is in production.

4.5.9 Message Oriented Middleware

Message Oriented Middleware (MOM) provides a mechanism of communication between applications without being coupled. The applications can talk to each other via

the MOM service provider without even knowing what is on the other side.

The key goal of messaging middleware is to ensure that producers and consumers do not have to bother about each others existence as long as they know their expected messages are delivered or received, respectively.

The MOM makes sure that all the consumers receive their messages while at the same time that all service providers publish their messages.

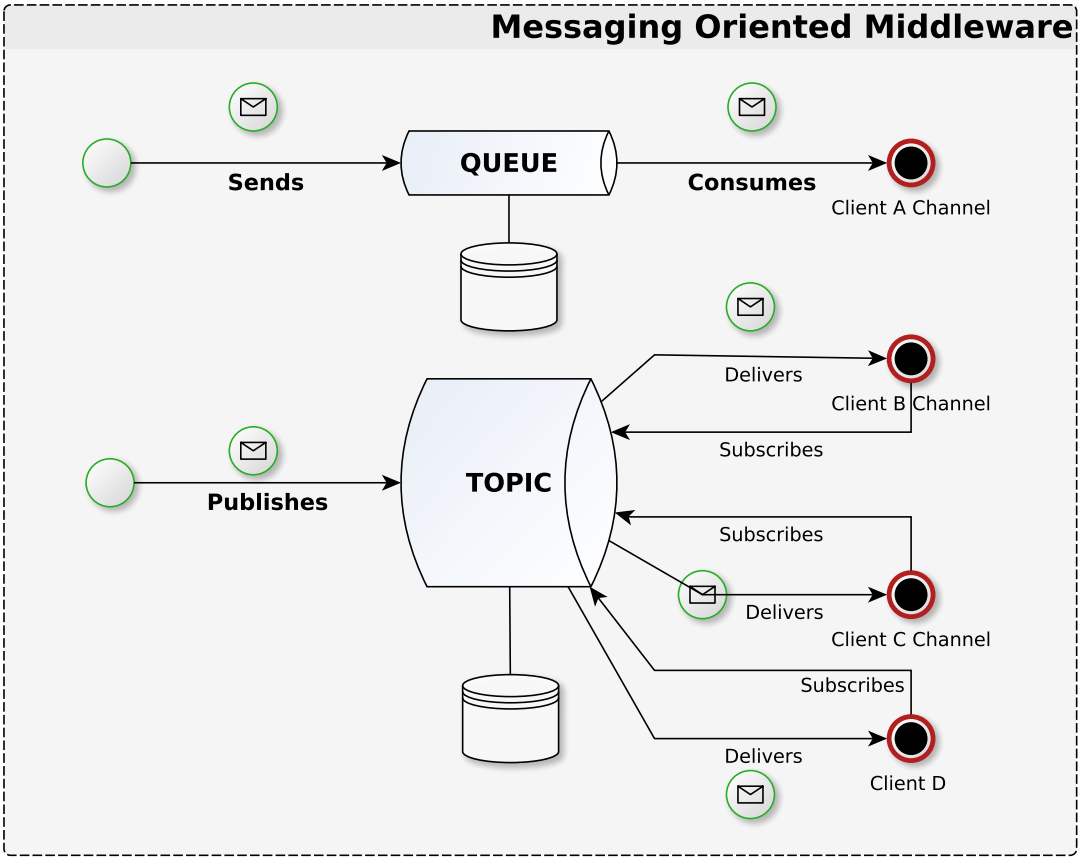

FIGURE 4.9: Message Oriented Middleware

There are primarily two messaging models in the architecture called point-to-point and publish-subscribe.

In point-to-point configuration, a message is delivered to a single consumer via a queue. A publisher publishes a message onto a queue while a consumer consumes the message off the queue. When a message is published to a queue, only one consumer can receive the message.

In a publish-subscribe messaging system, a message is published to a topic. Unlike the point to point configuration, there would be several subscribers each receiving a copy of the message.

The publisher publishes the message once to a topic. The consumers interested will subscribe to the same topic to consume that message.

The use of a MOM in this library is to guarantee message delivery by having local database storage in a MOM to store the message in case of communication problems and also cope with scalability issues that are likely to arise as the number of clients increase.

4.6 The Overall Architecture and Flow of Information in the exchange

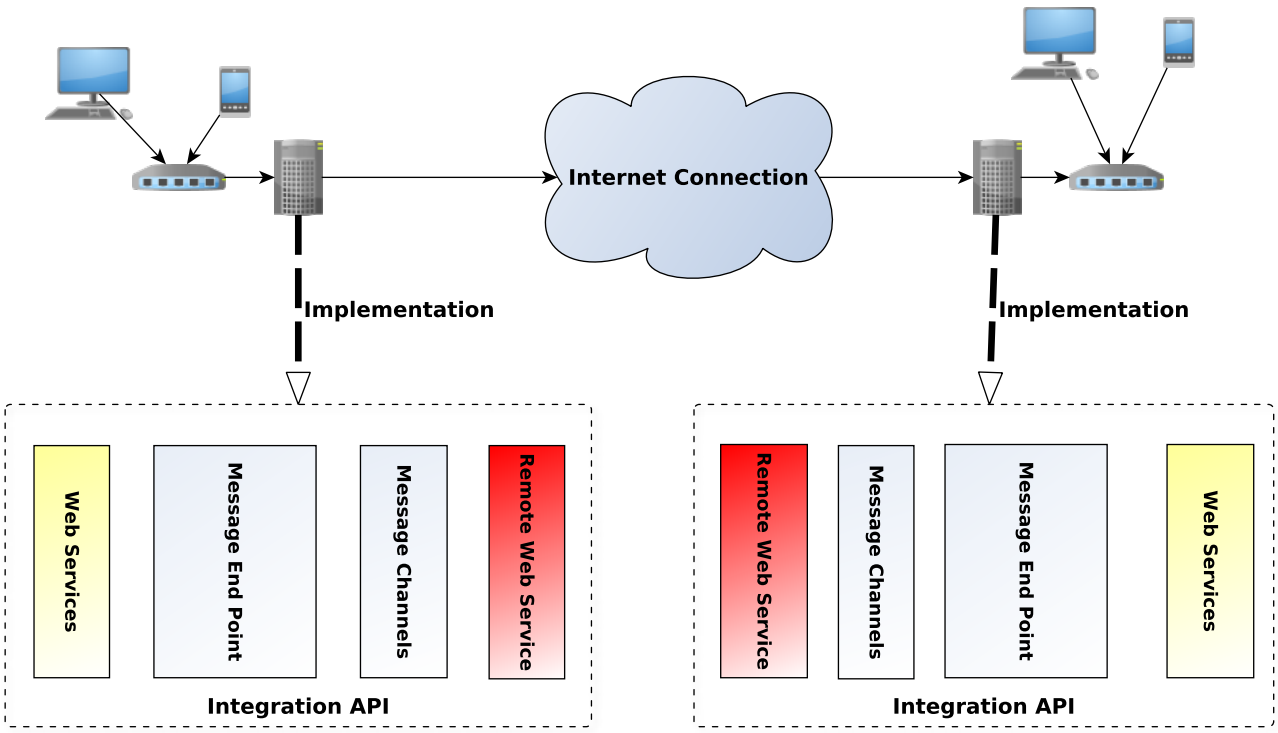

FIGURE 4.10: The Deployment Configuration of the Architecture

The landscape of health information systems deployed and being built in Africa continue to evolve. New systems being built either involve automating an existing process or

solving a new problem with a solution which allows a health institution to use the stored information in a way that was not possible before.

In some cases, the solutions consist of newly developed components that re-use already existing applications by delegating functionality to them, particularly legacy applications that implement complex logic and for which a complete re-write would be an unjustifiable cost. Other systems are divided from the beginning into multiple components that run independently to get the most out of the modern hardware and its high concurrency capabilities.

What is common about all these developments is that they tie together separate components and applications sometimes even located on different machines. One of the most important consequences of decomposing applications into multiple components is that these components can be expected to evolve independently.

To achieve this independent evolution of the separate systems, it is important to guarantee three things. First, loose coupling of systems. Secondly service based design and lastly reliable message passing or communication between the system.

Additionally, to make the systems extensible and scalable at runtime, the integration library has to be be made in such a way that the key moving parts of the architecture can be configured while the application is in production.

From the figure above, the integration library has both the local and remote web service API to enforce the service based design which imposes a service oriented architecture. The end points use the messaging system to transfer, convert and route messages in order to facilitate systems compatibility and low system coupling. The API has a message middleware component to ensure that messages are reliably delivered to the destination.

What is core and unique about the implementation of the mediation layer is that the routing rules and the transformation rules are stored in a data store and the end points read their configuration values from an embedded data store. With an embedded data store the behaviour of the integration library can be changed in real time to meet any need that arises. This makes the application built this way agnostic to evolving message standards and formats. Traditional integration libraries found in application like Apache

Mule, OpenESB, Apache ServiceMix and Spring Integration store routing information in XML files which are compiled with the deployed applications.

Chapter 5

Empirical Cases

5.1 Introduction

This chapter presents the three cases where some aspects of the solutions outlined in the previous chapter were applied. The first case involved the integration of two systems so that they can interoperate with each other. The second case involved the upgrading of an existing system so that it can be used in an environment where it will interoperate with systems that conform to the standards on the market and even to those that will be deployed in future, without having to rewrite the entire application. The last case involves the creation of a sample application that conforms and deals with existing interoperability issues, including any future changes that might arise. This case is specifically for a green field project being started from scratch.

5.2 Case Study I: Integration of Two Existing Systems

The first case study involves a project funded by the United States (US) President's Emergency Plan for AIDS Relief (PEPFAR) in Eastern Cape province in South Africa. The goal of the project was to improve access to quality HIV/AIDS, Tuberculosis (TB), Sexually Transmitted Infections (STI) prevention, care and treatment in the Eastern Cape Province. The Project is run by the Eastern Cape Regional Training Centre (ECRTC), a department at Walter Sisulu University (WSU) in the Faculty of Health Sciences.

This case was chosen based on these two conditions:

- The system under study should be one used for operation in a live situation. Live situation is meant to refer to an application that is being used in the day to day operation of an institution or organisation.
- Where integration of two or more system is involved, any system earmarked for retirement or cannibalisation was not to used in this study because the aim was to bring integration to existing systems while preserving the original investment.

5.2.1 Introduction

The Eastern Cape Regional Training Centre (ECRTC) is a project within the Faculty of Health Sciences at Walter Sisulu University (WSU) and was established in 2004 through a service agreement between the Eastern Cape Department of Health (ECDOH) and Walter Sisulu University (WSU) to provide, among other things, ongoing training to Health Care Workers (HCW) on HIV, AIDS, STI and TB in order to increase access and improve the quality of HIV, AIDS, STI, TB care and treatment.

- ECRTC provides technical assistance to ECDOH regarding the expansion of its HIV intervention programs and supports hospital and clinic site readiness for accreditation to provide comprehensive HIV care and treatment.
- ECRTC, through ongoing on site mentoring, has evaluated and demonstrated the HIV, TB and STI continuum of prevention, care and treatment practices in selected facilities; provided systems improvement support for sites to provide comprehensive HIV care including Anti Retroviral Therapy (ART), direct patient care and the opportunity for HCW to receive practical training.

Since April 2008 to March 2009 the accomplishments of ECRTC include, but not limited to the following: developing 3 training modules, preparing 15 hospitals and clinics for ARV accreditation; strengthening the capacity of 3579 Health Care Workers (HCW), including on site mentoring of Doctors (218), Nurses (2382), Other Facility Managers (979), Social Workers, Lay Counsellors and Community Health Workers(CHW) and provide mentoring to strengthen the provision of quality Anti Retroviral Treatment (ART) to

152 facilities with systems performance improvement for palliative care, PMTCT, Laboratory Services to strengthen and assure the quality of on-site nurse training on rapid testing, new tests such as polymerase chain reactions from dried blood spots, and specimen collections; and pharmacovigilance program to support and monitor emerging ARV and other drug-drug interactions and side effects; developing/implementing community prevention programs; and enhancing the capacity of NGOs to train and manage community health workers.

With the above facilities, ECRTC has invested in IT systems to collect data from these facilities to help with the operational management. This study specifically looked at the management of HIV/AIDS patients across ECRTC facilities.

Management of patients in these facilities is achieved using two systems.

1. ART Medical System
2. Intelligent Dispensing of ART system

5.2.2 ART Medical System

ART Medical System (ARTMS) is largely based on the OpenEMR, a free and open source electronic health records and medical practice management. OpenEMR is a complete ambulatory EHR certified by ICSA Labs, which deemed the system to have attained Complete Ambulatory EHR certification following rigorous and thorough testing, and is one of the most popular open source electronic medical records in use today. OpenEMR is supported by a strong community of volunteers and professionals all with the common goal of making OpenEMR a superior alternative to its proprietary counterparts.

ARTMS was forked from OpenEMR because of its open source licence. Open source software is software whose source code is published and made available to the public, enabling anyone to copy, modify and redistribute the source code without paying royalty fees. ARTMS was largely modified to conform with the use cases involved in managing an HIV/AIDS patient over a life time. The database was ported from MySQl to couchDB which supported offline disconnected nature and allowed synchronisation with easy conflict resolutions. A number of doctors take this application around the clinics in

the province on their computer laptops and only synchronise with the central master database when they report back to base.

Some of the features adopted and improvised from the OpenEMR include patient demographics, patient scheduling, electronic medical records, patient portal and reports.

Patient demographics is used for keeping and tracking information such as patient name, date of birth, sex, identification, marital status, contact information language, race and information about a patient supporter involved in the managing of a patient.

Patient scheduling is used for supporting multiple facilities spread across Eastern Cape to monitor patients movement in order to prevent patients from moving around without appointments. Scheduling is also used to schedule patient's appointments. Notification of schedules and appointment is done through social workers and patient supporters.

Electronic medical record is used for keeping patient encounters and any observation, recommendations that the medical personnel makes when the patient visits a facility.

Patient portal is used to see the status of the patient information and progress and compliance they are making or not making.

The system has the reports feature to enable the facility generate reports that are sent to ECRTC, provincial government and national government.

The ARTMS does not have a pharmacy module and the pharmacy module from the OpenEMR was not well developed at the time to deal with the pharmacy needs of up referral and down referral involved in rural clinics in Eastern Cape.

Rural clinics are organised around a main clinic , which has better facilities. The main clinic has small cluster clinics called down referrals. If the cluster clinic runs out of drugs, patients can be referred to the main clinic and the process is what is called an up referral. If the cluster clinic has drugs, the main clinics can refer the patient down to the cluster clinic and that process is know as the down referral. The key is to make sure that there is no duplication of patients data during the up and down movement of patients.

5.2.3 Intelligent Dispensing of ART

The Intelligent Dispensing of ART system (iDART) is a pharmacy application that allows dispensing of drugs both on site and from a remote pharmacy.

iDART was developed by Cell-Life in collaboration with the Desmond Tutu HIV Foundation (DTHF), Cape Peninsula University of Technology and University of Cape Town, as a software solution designed to support the dispensing of ARV drugs in the public health sector.

iDART system has two methods of dispensing. This is done by either creating packages for patients who are scheduled to visit the pharmacy on a given date to collect the drugs or directly dispensing to patients when they present themselves at the facility.

The key feature that made ECRTC install iDART was its remote dispensing capability. At the central dispensing pharmacy, the pharmacist creates a physical package consisting of ARV drugs for each patient enrolled in the treatment program. Upon completion, the packages are sent to the remote ARV site where ART patients visit regularly for medical consultations and drugs collection.

When a patient receives the packaged drugs, the nurse scans out the package using a hand held barcode scanner, thus allowing patients to receive treatment from clinics closer to their homes.

This system integrated well into ECRTC operations, which had a lot of rural clinics spread across the Eastern cape Province.

Operationally, the iDART system has features for tracking patient treatment and providing comprehensive patient treatment history. The system also helps in accurate Drugs stock control management to enable a facility manage its stock levels by incorporating a pharmacy management stock control, drug deliveries and drug-dispensing tools which allow a central pharmacist to provide services to multiple remote clinics.

5.2.4 Integration of the two systems

The two systems are used for specific needs. ARTMS is as a primary data store for patients records and has a module that social workers not only use use to track patients who have missed appointments, but also patients who are not adhering to drug regimen. Further, health workers also use ARTMS to enrol people into ART programs and assign initial regimen for patients. They base their decision on data that reside in the ARTMS.

iDART is only used in the pharmacy because the key component of managing HIV/AIDS patients over a long period is the dispensing of the drug and monitoring the compliance. Therefore, the appointments of patients who are already enrolled into the ART programme is managed by iDART. iDART also keeps track of the pill count, an indicator used to measure a patient compliance in taking drugs.

Because of the specific functionality of these two systems, there are 3 areas that emerged as areas of integration:

1. The ARTMS system needed to pass information of a patient with their regimen that the health worker has prescribed to iDART for enrolment into the iDART system.
2. The iDART system needed to pass information of the appointments made by patients to the ARTMS, which is used by the doctor to monitor any side effects and if possible change the regimen and pass it back again to the iDART system.
3. The iDART system needed to pass all the patient's visits to the pharmacy and the pill count statics to the module on ARTMS used by the social workers to track patients and help with compliant issues.

At the time this case was being worked on, there was so much information about SOA based integration of two disparate systems. Integration was done at the database level because there was poor API support for PHP based SOAP API. It was thought practical to do integration based on the databases of the two systems which were both based on the SQL language (PostgresQL for iDART and MySQL for ARTMS). The strategy was to build web services API around the databases and expose them to each other.

Patient Enrolment into iDART

The business process of managing an HIV/AID patient requires that a patient goes through a number of stages which include input from various sources before a patient can be enrolled onto the drugs program. The final stage of this process is the prescription of a particular regimen by a doctor and recording the date the patient has started taking drugs. This information is captured in ARTMS system. For iDART to start dispensing drugs, it needs a patient demographics and the regimen the patient has been put on.

This integration was solved by exposing the database table API that held a patient information and the regimen the patient put on. The interface was exposed as a web service API that offered the service with three key services. The first service was for creating a patient record on iDART. The second service was for adding patient's regimen details. Modifying the patient's current regimen was the third service.

FIGURE 5.1: Concept Model of Patient and Drug Regimen

Design on the iDART system involved creating domain objects that reflected the database snapshot of a patient and regimen record. The two domain objects were annotated with appropriate tags so that they can marshall XML or JSON datatypes.

With this web service interface in place, all the information the client needed was to make the appropriate calls to iDART and use the service that iDART exposed.

Modifications made to the ARTMS was to include the call to iDART whenever the patient was enrolled into the ART program and also make another call to iDART whenever a patient's regimen was changed by the doctor.

Patients Appointments into ARTMS

ARTMS does manage the patient appointments for all patients at various stages of of the ART program. However, once the patient is enrolled onto the program, the appointments are managed by the pharmacist using iDART. The pharmacist set the patient's next appointments. Every time a patient comes into the facility, they need to see the doctor to monitor their progress. The doctors can only see the patient's appointment using ARTMS. Therefore, a web service had to be created to expose the patient's record and appointment so that iDART could easily post the appointments to ARTMS.

The business process of making appointments on iDART was modified to make sure that a call is made to ARTMS whenever an appointment is made.

FIGURE 5.2: Conceptual Model of The Patient Appointment

Patients Drugs Compliance into ARTMS

The ARTMS has a module used by social workers or supporters to monitor the status of the patient. The social worker monitors things like pill count and whether the patient did show up for appointment or not. If there is an irregular pattern on this information or

if the patient fails to show up for appointment, it is the job of the social worker to retrieve the details of the patient and make a home visit to find out what could be wrong.

When the patient shows up at the pharmacy, the pharmacist needs to record the number of pills they were given and the number of pills that they have returned. This is recorded as pill count. Also, at the beginning of the day, the pharmacist has a list of people that are scheduled to come and collect drugs. As they collect the drugs, the pharmacist marks them as having come. At the end of the day, a list of those who did not show up is generated.

To deal with this scenario, a web service was created on iDART to expose a patients pill count record and a list of patients who did not show up on a particular day.

Changes made to the ARTMS module for social workers was to generate the list of patients who missed appointments and get the list of patients pill count and do the processing locally on ARTMS to see any irregular patterns in pill counts.

5.3 Case Study II : Upgrading an Existing System

The case is meant to demonstrate that certain applications can be enhanced to meet the interoperability needs required in an environment that require applications to collaborate. This case represent most of the applications that are built across Africa. The primary goal of this type of applications is to solve the problem at hand rather than be compliant with any standard.

5.3.1 Introduction

The case involves an application called Care Data App (CDA) developed in the incubation hub at Cape Peninsula University Technology to manage home based care patients. The need for the application came about after it was realised that the paper based system used by home based health care workers was slow, difficult to administer, difficult to track, difficult to store documents and manage the ever increasing number of paper that was being generated. Generating reports from the paper based system was also becoming increasingly difficult.

The business process of home based care starts with a nurse at the central location who manages the home based care workers that are deployed to various homes in the community to take care of patients. The home based care workers report to the central location where the nurse assigns them patients to see and gives them forms with a list of activities they must do while they are visiting patients and record whatever they have done, including any new incidences. At the end of the day the home based caregivers report back to the central location where they submit their reports to the nurse. Each health care worker is assigned a number of patients they have to visit in a community.

It was felt that there was a need to develop an application that could take advantage of the mobile network infrastructure and the cell phones the health workers had. The goal was to automate the home based care process and improve the data collection, storage and analysis.

5.3.2 Care Data Application

Care Data Application (CDA) is an application based on modern Java Enterprise Edition technologies that provide a framework to build scalable applications based on multi-tier architecture. CDA is based on PostgresQL as the data store, Springframework as the provider of enterprise features such as transactions , security, naming, Object Relational Mapping (ORM) management using Hibernate Java Persistence API (JPA) and many other features that promote abstractions and separation of concerns.

Architecturally, the application is divided into four tiers as shown in Figure 5.3. The data layer is responsible for ORM management, the service layer is responsible for any business processing. The service layer has an interface that is used by the desktop client and the REST based web services layer.

The desktop layer is used to provide a web based or browser window into the application. The browser based window is used by the nurse and the system administrator to manage the application. The nurse uses it to add patients and their care plans to the system and then assign them to health care workers on the system. The administrator adds the nurses and health care workers to the system and sets the appropriate permission roles for each user of the system.

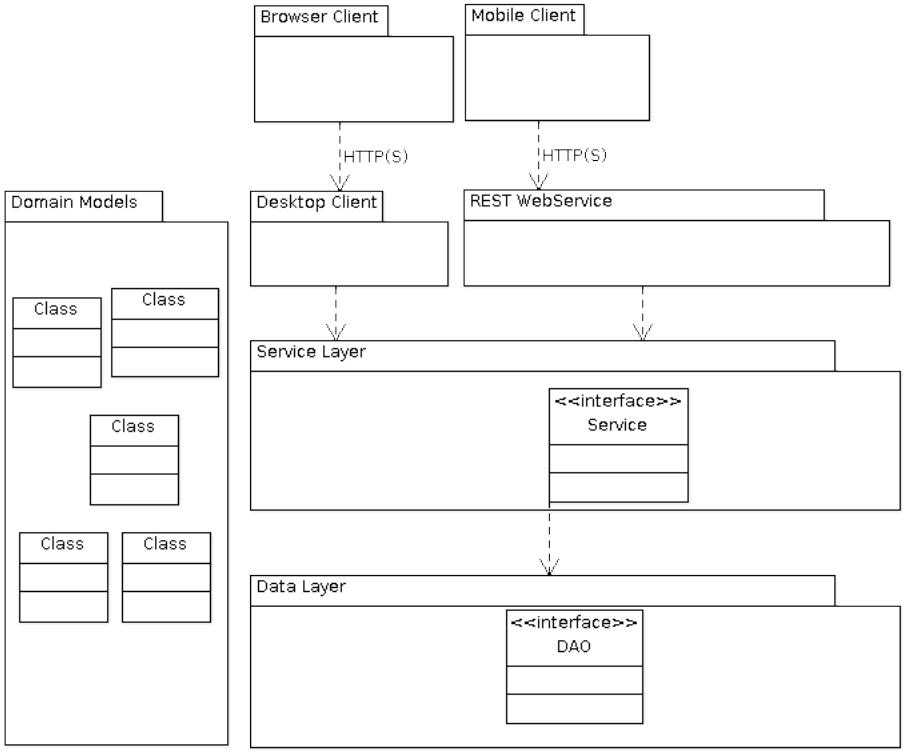

FIGURE 5.3: Model Architecture of CDA minus cross cutting concerns like security

The web services component exposes a JSON based web service API that is used by an application that runs on a health worker cell phone to collect data and pass it back to the server.

The application that runs on a mobile cell phone is based on Java Micro Edition (JME) because it is the technology that is supported by the cheaper and ubiquitous phones that most of the home based care givers use.

Operationally, the nurse selects the patients and assigns them to a care giver. Every morning, the care giver launches the application installed on their phone to download their schedule from the server. Their schedule includes a list of patients they have to visit on that day and the tasks they have to do. With the list on their phone, the health care workers visit the patients, checks off all the tasks and submits the results back to the central server. At the central location, the nurse can check and see which patients were visited and what action was taken.

Please note that both the browser client and the mobile client runs outsides the space where the server is running and communication is via HTTP protocol. The domain models have cross cutting concerns and cuts across all the layers where they are used.

5.3.3 Integration module

CDA development was based on solid Object Oriented Principles (OOP) that provided the architecture with extension points on the service layer. So the enhancement of CDA tapped into the REST based web services layer as the point of contact for the integration layer as shown in Figure 5.4.

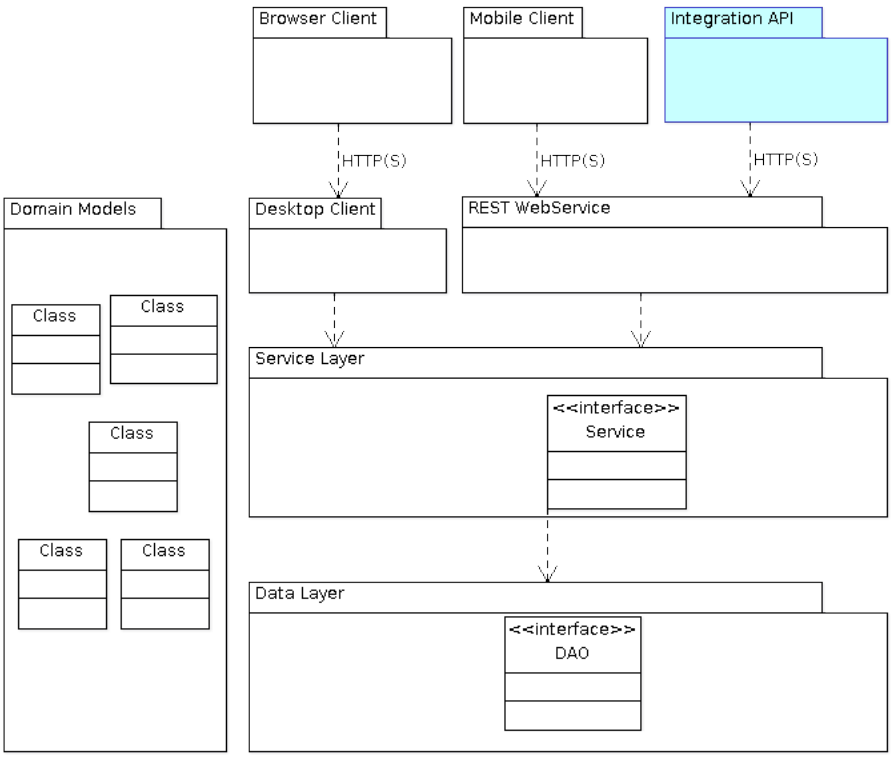

FIGURE 5.4: CDA Architecture with the Integration Library shown

All data going in and out of the REST based web service layer fed into the integration library, which passed all messages through layer for conversion to any standard required.

There were two open source integration libraries on the market for use in the solution to create ends points and process the messages. The integration libraries are Spring Integration and Apache Camel. Spring Integration was evaluated and discarded because of its lack of support for many connection protocols and its relatively newness to the integration space. Apache Camel was chosen because of its longevity and flexibility that allowed the use of custom processes and processing of custom dynamic routes required in this work.

To describe the integration in detail, we shall show how a message with a patient details, generated by the CDA, is converted into an HL7 message and passed on to the consumer who needs to consume an HL7 message.

To convert a message to HL7, there are three pieces that had to be in place. First, the reading of the message from the web service end point and converting it into a Java object. Secondly, the conversion of the Java based message into the HL7 format. Finally, the publishing of the HL7 message to the end point.

The integration library reads from the web service API that is producing JSON formatted messages, which is also used by the mobile client. The listing shows the conversion of the message to a Java object. The JSON format is passed to the **MessageProcessor** to convert the message into the Java object, which is then marshalled into an HL7 messages and passed to the queue end point, **hl7Channel**, where it can be consumed by other clients.

The Java object is then fed into another converter shown below that now produces an HL7 message. The HL7 message is now sent out to the output

```
1  import org.apache.camel.builder.RouteBuilder;
2  import org.apache.camel.component.hl7.HL7DataFormat;
3  import org.apache.camel.spi.DataFormat;
4  import zm.hashcode.hashmed.processors.MessageProcessor;
5  public class HL7Route extends RouteBuilder {
6      DataFormat hl7message = new HL7DataFormat();
7      @Override
8      public void configure() throws Exception {
9          from("https:localthost:8084/caredata/ws/patient/")
10                 .process(new MessageProcessor())
11                 .marshal(hl7message)
```

```
12                          .to("jms:queue:hl7channel");
13          }
14  }
```

There is also a similar reverse channel for messages that are coming into the system for conversion into JSON format and fed into the system.

The CDA has mostly been used to produce HL7v2 messages. Using the same processes other message production and consumption can easily be implemented.

5.4 Case Study III: Creating a new System

The last case involved looking at building an application from scratch that would address all the issues that are encountered in the health care domain. There are many applications across Africa that continue being build from scratch to address different domains like laboratory, pharmacy, patient admission and many other aspects specific to the health care space.

There has been a lot of work and research that has gone into creating a base health care information domain framework that could ease the creation and standardisation of applications in the health care domain. Key among these health care information domain are driven by the HL7 standards group and the OpenEHR group.

The HL7 standard group have produced what is called the Reference Information Model (RIM), which can be used as a base model framework to create a model that can be used to produce applications that conform to standards based on HL7 V3. The group has even produced a Java implementation of RIM.

The OpenEHR group has produced what they call Archetype Driven Models. Archetypes, which are constraint-based models of domain entities, describes configurations of data instances whose classes are described in a reference model.

The OpenEHR group have produced reference implementation of the openEHR reference model and the openEHR Archetype model that can be used to build applications.

Both HL7 group and the OpenEHR have solid implementation base to build on. In this work we chose to use the HL7 RIM model as a base to build an application from scratch.

5.4.1 Introduction

HL7 is an organisation which has interest in the development and advancement of clinical and administrative standards for health care. It is one of several Standards Development Organisations (SDOs) accredited by the American National Standards Institute (ANSI).

HL7 develops specifications, with the most widely used being a messaging standard that enables disparate health care applications to exchange key sets of data. As a specification, the organisation has developed RIM as a static model of health care, which is a combined consensus view of information from the perspective of the HL7 working group and the HL7 international affiliates. The RIM is expressed in UML with HL7 specific tags as extensions to the UML model element metadata.

The application we set out to build from scratch was called Hashmed and was to use HL7 V3 RIM as the basis for creating any health information system. We looked at the Java implementation of the RIM which looked abandoned but had the necessary code and good guidance documentation to get started.

There are guidelines to be followed when building applications that are compliant with HL7 version 3. The RIM is the basis for all applications that are compliant with version 3. The RIM is domain agnostic and stable. From the RIM several domain models for applications can be built based on a specific domain like pharmacy, laboratory or clinical systems using a subset model called the Domain Information Model(DIM).

From the DIM a messaging model can created by selecting the Models that are specific in order to create a Domain Message Information Model (D-MIM). The D-MIM is refined further to create a Refined Message Information Model (R-MIM). From the R-MIM, the Interaction Model and the Hierarchical Message Description (HMD) are created. The final step in the process is the creation of the Implementation Technology Specification.

5.4.2 Hashmed

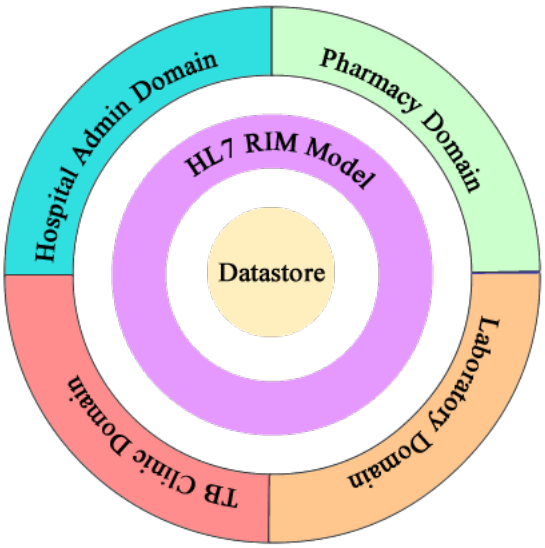

FIGURE 5.5: The architecture of Hashmed

Hashmed is a framework stack that can be used to build new health information systems for any domain in this space. The first task in building the stack was to map the entire HL7 RIM classes, including their data types to a Java JVM programming language software domain that is persisted into a document database because of the excellent replication and performance in disconnected environment.

These models were exposed as building blocks for a constrained domain specific application, such as pharmacy, which is created from a use case model to produce a D-MIM.The Messaging model was also created based on the persisted model to transport messages to a mediation layer.

Figure 5.5 shows the architecture of Hashmed where data to be stored moves from the outer ring consisting of D-MIM and R-MIM for specific domain areas like pharmacy, laboratory, clinic and hospital administration. This layer is made based on the core HL7 RIM model which is persisted in the Data store using object document mapping technology to convert object domain models into document files used by the NoSQL database called CouchDB. The object document mapping is responsible for marshalling back and forth the conversion between the RIM objects and the documents stored in the data store.

5.4.3 Interoprability with Future Apps

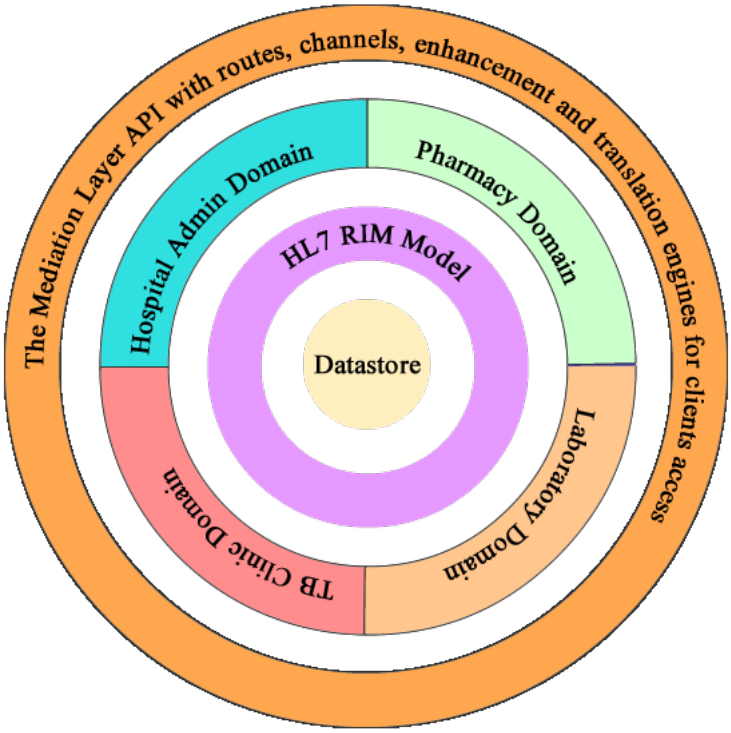

FIGURE 5.6: The Mediation layer

Interoperability and insurance against future standards in Hashmed is provided through a mediation layer show in Figure 5.6. The mediation layer is built around a pluggable routing engine that reads information from the data store for both incoming and outgoing destinations. Reading decisions from the data store enable the system behaviour to be changed in real time. Routes information can be dynamically added to the system. All clients write their method calls to the API exposed by the mediation layer.

Chapter 6

Discussion of Results

In this chapter we look at the results of the cases listed in the previous chapter and describe what was learnt during the development and deployment. The evaluation of the artefact produced is also covered in this chapter. The last section looks at the the contributions of this work.

6.1 Challenges and Lessons Learnt

6.1.1 Integration of Two Existing Systems

Integration in this case was based on wrapping web services API around two databases and exposing them to each other. The first cut of the integration was done using SOAP based web services. Initially, it was done by using tools to reverse engineer the databases involved and expose them as SOAP based web services interface. Heavy tooling was used to quickly generate the server side proxies, marshallers and servlet classes required to expose a SOAP based web service.

However, this become very problematic as the changes at the early stage were getting frequent and fine tuning the application meant having to work with re-generated code which was littered with a lot of verbose non functional lines of code. This significantly slowed down the testing and deployment of the solution.

Because of this, the integration using SOAP interface had to be abandoned for the REST based solution.

It must be noted that the changes to both systems was possible because the source code repository tree for both products, including their data models, were freely available.

Key lesson learnt was that, at the start of building an application, it is vital to use technologies that allow quick development, testing and deployment in order to deal with the high frequency of requirements change. Another lesson learnt was that this type of integration was quicker if applications to be integrated are based on relational databases.

6.1.2 Upgrading an existing System

Integration with a mediation layer in this application did not present a great challenge because the architecture of the application was well partitioned, with well documented and modularised API for extension points.

However, the major challenge was dealing with the ever changing routes, destination and nature of end points. The initial bulk of the integration configuration and routing was done in static XML files. This led to a situation where the application had to be stopped, recompiled and re-deployed to make any new changes available to the clients API.

Further, more work had to be done in consuming incoming messages from some systems. Not much was tested on health care standards as it turned out that most of the clients who used or wanted to use CDA app did not really care about the standards. All they wanted was a functional system that could solve their immediate problems.

Key take away from this case was that systems in this age should always be designed with extension points in mind as this could significantly reduce the amount of risk and work involved in scaling the applications to deal with new challenges.

Additionally, what was learnt was that the pervasive static XML configuration in most modern application is not the best method to use if one wants to build a system that is adaptive at runtime.

6.1.3 Creating a new System

This architecture was built around a new and evolving technology, particularly the object document mapping drivers and the underlying database based on the new document based NoSQL database. Since the technologies are new and in constant development, the API was not stable. The application kept breaking with every release of the new version of the database and the drivers for object relational mapping.

The last two years have seen the stabilisation of CouchDB, the storage engine used and the API used for object document mapping. Key lesson learnt in this process was that it was wrong to hard wire the routes and mediation decisions into the mediation layer. While this solved specific cases, it always created more work and delays if one had to change the decisions or routes.

The good side of this case was the realisation that it was possible to upgrade the mediation layer so that the key aspects that are subject to constant change and revision can be abstracted away into a location where they could be dynamically introduced into the application. This is what led to the creation of a service that stores routes and decisions into a database so that the routes can be changed at runtime without breaking or bring the application in production down.

6.2 Assessing the Artefacts Produced

This section assesses the results produced in this work. The discussion will look at three aspects of the work done.

First, the review and assessment of the artefacts produced by this research. The artefacts produced in this research are the mediation API, the design methodology, other related constructs and methods.

Secondly, the evaluation of the research conducted in line with the constructive science and natural science will be assessed.

Lastly, the research methodology that has been adopted to produce the artefacts will be evaluated too.

6.2.1 The Design of the Artefacts

Assessing the designing of software artefact is largely based on the knowledge mined by developers over the years.

In their work, (Chahal and Singh, 2009) looked at what metrics are used to assess the quality of a software product. They highlighted object oriented design characteristics such as abstraction, coupling, cohesion, inheritance, polymorphism and encapsulation. Good management of these characteristics leads to software artefacts that leads to early indication of the goodness of software design.

As software is being developed, the (Martin, 2003) in his book listed Rigidity (Design is difficult to change), Fragility (Design is easy to break.), Immobility(Design is difficult to reuse), Viscosity(It is difficult to do the right thing), Needless Complexity(Overdesign), Needless Repetition (Cut and Paste Mouse abuse) and Opacity (Disorganised expression) as characteristics that should not be allowed to manifest in the life of the artefact under development.

In the book, he described how to recognise these characteristics and architect designs that avoid these bad attributes.

Since the artefact was developed with the help of students, certain design guidelines that took care of the issues raised above were put in place and constantly monitored to make sure students strictly adhered to guidelines in order to ensure the production of quality designs.

All the boundary APIs that exposed both intra and inter module services were coded to an interface and the implementation hidden away in conformity with the Dependency Inversion Principle (DIP) (Nordberg III, 2001) which requires dependencies to be coupled to interface abstractions and not concrete details. Using the an Inversion of Control (IoC) framework, the interface and implementation were successfully designed to vary independently. This enabled the design to address issues like abstraction, encapsulation of modules, rigidity and fragility.

All the naming convention in the API was been strictly used to reflect the names that are in the domain names. Meaningless names for variables, methods and modules like “x” were not allowed throughout the API. Methods length where capped at a maximum

of 15 lines of code. This helped the API deal with the issue of opacity as the application passed from one set of developers to the others. New developers were able to make sense of meaningful names. The short methods made it easier for new developers to understand the application logic and be up to speed quickly with the rest of the team.

To further deal with constant change, the Open Closed Principle (OCP) (Martin, 1996), which requires artefacts to be closed for modifications and open for extensions was enforced throughout the design. Once a part of the system was developed and tested, it was closed to any modification, but an interface was provided using software design patterns to enable the behaviour of the API to be extended. This not only helped further with dealing with fragility, but also helped with immobility and viscosity.

Packaging of classes and modules was designed in such a way that communication between modules was lessened. This was achieved by making sure that modules that have similar characteristics in terms of purpose and change were grouped together. Basically low coupling and high cohesion was the goal of packaging.

So, all knowledge that is well known and documented was used in the design of the mediation API to ensure that we produced a good quality designed API that can easily be extended and re-used with ease.

6.2.2 The Design Method Process

The ingredients that goes into the development of the quality software artefact has been well known for many years and widely documented (Conradi and Westfechtel, 1998). The environment was set up based on the industry recommended process of building large scale software.

To stick to these values, the version control system called subversion was put in place to ensure that the whole software development process was well tracked and given room to evolve. The group would check in the code they have worked on and check out the code to work with.

The development process of the artefact followed the Test-Driven Development(TDD) process that is widely used in the development of software artefacts and well documented by (Beck, 2003). TDD is the art of building software artefacts by specifying a requirement through a “test before writing production code” method.

This process was used so that the needs of the mediation API needed to do were defined before coding them down in a program. This methodology gave us the idea of what the API should look like and be used for before it was built.

The process started with tests, each meant to define a single purpose such as “Produce patient demographic data that is HL7 V3 in json format”, “Save the Patient Data into the CouchDB database,”

The programmers wrote the test code as if they were developers using the API. Some of the key questions they kept in mind included considering how a user would instantiate the API, what problems would a user encounter by calling the service and what errors or exceptions should they expect?

The motto was add the test, add the method signature without the implementation, make the test fail, and then add the production code to satisfy the test. After the initial test and production code produce successful results, the next step was clean up the production code to get rid of unnecessary code generated in the process of making the test pass.

The mediation API was built using 3 groups of students over a period of three years as sub projects for their real world assignments. Therefore, rules had to be established from the outset about the build and verification process of what was being contributed to the mediation API. Key to the verification process was building the four layers of testing namely unit, mocking, integration and regression tests.

Unit tests were used to test other external dependencies, like service calls against the data store. Mocks were used test code without actually making calls to real API or systems. Integration test was used when two interfaces were connected to work together. Regression test was run on all the code to ensure that any changes introduced to the code did not break what was working and this was accomplished using a test suite that ran all tests in the artefacts.

The key benefit that this process brought was the ability to assemble development teams separated over time and space to work independently on the same interface. Teams were able to agree on certain shared API and then go off on their own intensive paths where one team implements the interface and the other codes up objects that depend on the interface.

For example, two teams would decide on the methods needed for patient data access to the database, after which one team creates a data access implementation for a data store, while the other team worked on the business methods using the data interface without actually needing a complete data store API to do the work. One team working using Unit testing and the other using Mock testing.

When both teams were done, all the parts were then integrated together and tested as an integration test. Regression testing was performed successfully before the code was committed to the main code branch.

6.2.3 Design Evaluation

The artefacts have been theoretically and practically assessed. Cases were used as evaluation vehicles for utility and quality of the artefacts.

In case I, the REST web service API and the solution for disconnected architecture have been in production use for almost four years with success. Today, the two integrated applications continue to share the data. Doctors and nurses continue to carry the developed application on their laptops which they sync with the central database. The only upgrades done have been versioned software of the CouchDB database. Feedback and issues faced have led to the refinement the artefacts. Experiences gained from the first case led to the refined development of the mediation web services library.

Case II helped reveal the downside and limitations of the solutions of having to hard code routing and mediation information in the mediation library. The disconnected architecture capability was also tested and evaluated with the use of dumb cell phones in case II.

By adopting standards processes and recommendations used in the production of software artefacts, the quality of the library has been assured of devoid of common documented errors.

In terms of efficacy, some aspect of the design have changed the way people manage data in case I and enabled the organisation involved to continue using the investment they made in the initial application.

Further more, the lessons learnt in Case II have permanently changed the way routing and mediation rules are injected into the application design. Traditionally, this information has been kept in flat files that required rebuilding the application if change of the behaviour of the application is made.

6.3 Research Contributions

The problems as we have identified are relevant and well documented in the introduction section of this work. There is active work in many standards bodies to try and address the problems highlighted on a global scale.

The need for integrated health information systems continues to grow. More systems are being added to the ecosystem without adhering to a standard that is not currently in place. While most interoperability works at the physical layer of networks, interoperability at application layer, both syntactically and semantically has been a challenge in Africa due to the unreliability of the existing network connectivity.

The quality of the connectivity in Africa also has compounded the problem, making it hard to just get solutions from developed parts of the world and deploy them in a resource constrained environment.

In this research, we have tried to find solutions that recognise the African context and the fact that standardisation is a process that takes time to be effected. The problem solved is not only on how to make a SOA in Africa scale and work reliably, but also how to build architectures that can easily be upgraded or adapted to ever evolving standards.

The result of this research was the understanding of the key factors that have continued to be major challenges in this field. The key contribution is the mediation architectural

design that is able to not only deal with the mediation or interoperability issues, but also operate in an unreliable connectivity environment and be able to adjust to the ever changing and evolving Health Care Standards.

Disconnected architecture is achieved by the use of data stores that relies on data versions rather than availability of the network to operate. Data that determines the routes of the messages is also feed dynamically into the application to change the message flow of the application.

The translation of messages in the library had been traditionally done by wiring mediation information into the application. In this mediation library, the mediation information is fed from the data store. This makes it possible to add any translation data between any two end points without having to redo or rebuild the application.

6.3.1 Theoretical Contributions

There are three key theoretical contributions made by this work namely SOA Driven Disconnected Architecture (SOA-DDA), Runtime Dynamic Routing (RDR) and a methodology to enhance an application for future interoperability.

Theory behind disconnected architecture is based on the premise that if messages can be held locally in a versioned state, transportation and subsequent merging of like messages at the destination can take place with conflict that can be easily and humanly detected and corrected.

Connected architectures need the connection to ensure the integrity of the messages to be transported. The preservation and protection of mutable state of the messages being moved around is cardinal. If messages are versioned, this introduces immutability in the nature of messages moved. This in turn makes state preservation, with its connection oriented overheard, irrelevant. As a result of this scenario, conflict resolution in transported messages can be deferred to a later time whenever network connection is available.

It is a great challenge to develop a system based on an ever evolving standard. The key was to identify the moving parts in the health care domain that are highly susceptible to the volatility of standards used when negotiating message exchanges.

The runtime dynamic routing is based on the fact that if information that keeps routing intelligence can be fed into the system at runtime, it is possible to alter the behaviour of the system at runtime. This key point has been used in the design of abstracting the routing information and mediation data to external data store that can be changed based on the type and format of the messages to be exchanged.

Finally, a method has also been developed on how to enhance a system that is not inter operable and make it ready for interoperability.

6.3.2 Practical Contributions

In the search for theoretical contributions, the work had to produce software artefacts. The practical contributions is a SOA-DDA framework for building interoperable health information systems that can operate in resource constrained environment like Africa. The framework takes care of all the issues that need to be addressed in order to build a workable system.

While the framework might not be used in the already existing system, it is useful for systems that are being newly built. Further, an API based on HL7 RIM that could be used to build systems from the ground up has been produced. The API exposes REST based web services to the consumers of the services.

6.3.3 The Advantage of the Solutions

The core function performed by the solutions proposed in this work is routing, transformation and enhancement. These are features that any system trying to solve interoperability needs to address at both the syntactic level and the semantic level. The advantage that the designs described in this paper is the method in which information to effect the above changes is dynamically fed into the system and not physically wired. This creates the advantage of not only being able to change the system at run time, but also being able to provide the ability for the designs to adapt to the ever evolving standard without having to break down or rebuild the system from scratch. The system is simply standards agnostic.

The philosophy of coding to interfaces and coupling to only abstractions makes the design easy to use in assembling even more complex structures from the base API that is based on a solid HL7 RIM model. Furthermore the HL7 RIM model is abstracted away from the other layers and so can be easily replaced with other models like openEHR to achieve the appropriate desired effect.

With the work being premised in Africa, connectivity and availability issues are taken care of in these designs. The design has a disconnected architecture in mind and with the data store being CouchDB, which has been proved to work well in disconnected environment. The disconnected architectures solution has been successfully used in case I for the last 4 years and it has worked well and continue to be in use to this day.

All the end points of the API have web services interface based on REST. These interfaces enable the API to be part of a big SOA of any hospital enterprise being built. The API can be integrated into an already built system as described in Case II

6.4 Future Directions

The current solution of integration requires deep knowledge of both systems that need to be integrated. The routing and mediation knowledge loaded at runtime in the data store has to come from an expert who has deep insight into the knowledge of the semantics the two systems, which are to be integrated, use.

Further work is needed to come up with algorithms that are based on machine learning that could be used to guess or interpret the semantics that are contained in an incoming message and route the message to the correct route and interpretation engine. This is the natural course of action to enhance this work and reduce the amount of work and knowledge of integrating systems into the mediation API.

There is also the rise in popularity and usability of programming languages , such as Scala, Rust and Clojure, which by default, have immutability built into them. These languages provide other opportunities to enhance solutions that are more robust in disconnected and resource constrained environment.

Chapter 7

Conclusion

In this work, we set out to find answers to the increasing number of silo applications that are being developed and deployed across Africa. Our aim was to see if we could use Service Oriented Architectures (SOA) to address the problems of interoperability of silo systems deployed in Africa.

The literature survey done revealed a landscape that has a number of challenges one has to navigate in order to develop a solution that is sustainable in Africa. The landscape showed three types of silo systems shown in Figure 2.2 that one has to be aware of when working on a solution that addresses the architecture needs of the African environment, in addition to skills gap issues.

Additionally, the literature showed that architectures deployed in a resource constrained environment, like Africa, need to be able to operate in a disconnected environment.

From the problem framing and definition, it can be seen that although the problem was framed in the social science space, the solution required was an engineering one. The solution had a major influence on the research process chosen and used.

The mediation library was constructed using rigorous documented processes used to produce constructs, models, methods and an instatiation.

As an instatiation, the mediation architectural library is able to not only deal with the mediation and interoperability issues using web services standards, but also operate in

an environment where connectivity is unreliable. Furthermore, the library is able to adjust to the ever changing and evolving health care standards by providing a mechanism to update the mediation rules at runtime.

After the library was created, the output of the constructive science process was fed into the natural science process for theorising and justification in order to identify the practical and theoretical contributions of this work.

There are three key theoretical contributions made by this work namely SOA Driven Disconnected Architecture (SOA-DDA), Runtime Dynamic Routing (RDR) and a methodology to enhance an application for future interoperability.

The practical contributions is a SOA framework for building interoperable health information systems that can operate in resource constrained environment like Africa. The framework takes care of some of the issues that need to be addressed in order to build a workable system.

Appendix A

Appendix A

```
1  public interface PatientResource {
2  public abstract Collection<Patient> getPatient() ;
3      public abstract Patient getPatient(Long id);
4      public abstract Patient getPatient (@Context Request ↩
       request,RequestForm form) ;
5      public abstract Response createPatient (@Context Request↩
       request,PatientForm form);
6      public abstract Response updatePatient (@Context Request↩
       request,Patient patient);
7      public abstract Response updatePatient(@Context Request ↩
       request, String data);
8      public abstract Response updatePatientWound(@Context ↩
       Request request, String data);
9  }
```

Listing A.1: Interface for a Patient Resource

```
1  @Path("/patientservice")
2  @Produces({"application/json"})
3  @Service("patientResource")
4  public class PatientResourceImpl implements PatientResource ↩
       {
5
6      @Autowired
7      private PatientService patientService;
8
```

```
9       public PatientService getPatientService() {
10          return patientService;
11      }
12
13      public void setPatientService(PatientService ↩
       patientService) {
14          this.patientService = patientService;
15      }
16
17      @GET
18      @Path("/patient")
19      @Override
20      public Collection<Patient> getPatient() {
21          return patientService.findAll();
22      }
23
24      @GET
25      @Path("/patient/{id}")
26      @Override
27      public Patient getPatient(@PathParam("id") Long id) {
28          Patient patient = patientService.find(id);
29          if (patient == null) {
30              ResponseBuilder builder = Response.status(Status↩
       .BAD_REQUEST);
31              builder.type("application/json");
32              builder.entity("<error>Patient Not Found</error>↩
       ");
33              throw new WebApplicationException(builder.build↩
       ());
34          } else {
35              return patient;
36          }
37      }
38
39      @POST
40      @Path("patient")
41      @Override
```

```
42      public Patient getPatient(@Context Request request, ↩
       RequestForm form) {
43          try {
44              Patient patient = (Patient) patientService.↩
       getByPropertyName(form.getPropertyName(), form.↩
       getPropertyValue());
45              return patient;
46          } catch (PatientNotFoundException ex) {
47              ResponseBuilder builder = Response.status(Status↩
       .BAD_REQUEST);
48              builder.type("application/json");
49              builder.entity("<error>Invalid Code</error>");
50              Logger.getLogger(PatientResourceImpl.class.↩
       getName()).log(Level.SEVERE, null, ex);
51              throw new WebApplicationException(builder.build↩
       ());
52
53          }
54      }
55
56      @POST
57      @Path("patientss")
58      @Override
59      public Response createPatient(@Context Request request, ↩
       PatientForm form) {
60          Patient patient = new Patient();
61          patient.setPerson(form.getPerson());
62          patient.setUser(form.getUser());
63
64          try {
65              patientService.persist(patient);
66              ResponseBuilder builder = Response.created(new ↩
       URI("http://localhost:8084/hashmed/ws/patientservice/↩
       patientss"));
67
68              return builder.build();
69          } catch (URISyntaxException e) {
70              throw new RuntimeException(e);
```

```
71          }
72      }
73
74      @PUT
75      @Path("/patients/update")
76      @Override
77      public Response updatePatient(@Context Request request, ↩
    Patient patient) {
78          if (patient == null) {
79              return Response.status(Status.BAD_REQUEST).build↩
    ();
80          } else {
81              patientService.merge(patient);
82              return Response.ok(patient).build();
83          }
84      }
85
86      @POST
87      @Path("saveanswers")
88      @Override
89      public Response updatePatient(@Context Request request, ↩
    String data) {
90
91          try {
92              Patient patient = new Patient();
93              ConvertDate convert = new ConvertDate();
94              long patientId = 0;
95              JSONArray JSONCareActivityHistoryArray;
96
97              JSONCareActivityHistoryArray = new JSONArray(↩
    data);
98              if (JSONCareActivityHistoryArray.length() >= 0) ↩
    {
99                  JSONObject CAH_JSON = ↩
    JSONCareActivityHistoryArray.getJSONObject(0);
100                 patientId = CAH_JSON.getLong("patientId");
101                 patient = patientService.find(patientId);
102
```

```
103                 if (patient == null) {
104                     ResponseBuilder builder = Response.↩
    status(Status.BAD_REQUEST);
105                     builder.type("application/json");
106                     builder.entity("<error>Patient Not Found↩
    </error>");
107                     throw new WebApplicationException(↩
    builder.build());
108                 } else {
109                     for (int x = 0; x < ↩
    JSONCareActivityHistoryArray.length(); x++) {
110                         JSONObject CareActivityHistoryJSON =↩
     JSONCareActivityHistoryArray.getJSONObject(x);
111                         CareActivityHistory ↩
    CareActivityHistory = new CareActivityHistory();
112                         String visitDate = ↩
    CareActivityHistoryJSON.getString("visitDate");
113                          //TODO method commented cos for now↩
     on answers will besaved using visits and not patients
114                        // CareActivityHistory.setPatient(↩
    patient);
115                         CareActivityHistory.setVisitDate(↩
    convert.String2Date2(visitDate));
116                         CareActivityHistory.setCareActivity(↩
    CareActivityHistoryJSON.getString("careActivity"));
117                         CareActivityHistory.↩
    setActivityAnswer(CareActivityHistoryJSON.getString("↩
    activityAnswer"));
118 
119                         patient.getCareActivityHistory().add↩
    (CareActivityHistory);
120                     }
121 
122                     patientService.merge(patient);
123                     return Response.ok(patient).build();
124                 }
125             } else {
```

```
126                 ResponseBuilder builder = Response.status(↩
    Status.BAD_REQUEST);
127                 builder.type("application/json");
128                 builder.entity("<error>Patient Not Found</↩
    error>");
129                 return Response.status(Status.BAD_REQUEST).↩
    build();
130             }
131
132
133         } catch (JSONException ex) {
134             Logger.getLogger(PatientResourceImpl.class.↩
    getName()).log(Level.SEVERE, null, ex);
135             return Response.status(Status.BAD_REQUEST).build↩
    ();
136         }
137     }
138
139     @POST
140     @Path("savewound")
141     @Override
142     public Response updatePatientWound(@Context Request ↩
    request, String data) {
143         try {
144             Patient patient = new Patient();
145             ConvertDate convert = new ConvertDate();
146             long patientId = 0;
147             JSONObject woundJSON;
148             woundJSON = new JSONObject(data);
149             if (woundJSON.length() >= 0) {
150                 patientId = woundJSON.getLong("patientId");
151                 patient = patientService.find(patientId);
152
153                 if (patient == null) {
154                     ResponseBuilder builder = Response.↩
    status(Status.BAD_REQUEST);
155                     builder.type("application/json");
```

```
156                     builder.entity("<error>Patient Not Found</error>");
157                     throw new WebApplicationException(builder.build());
158                 } else {
159                         Wound wound = new Wound();
160                         wound.setWoundtype(woundJSON.getString("woundType"));
161                         wound.setColour(woundJSON.getString("colour"));
162                         wound.setSmell(woundJSON.getString("smell"));
163                         wound.setExudate(woundJSON.getString("exudate"));
164                         wound.setSiteArea(woundJSON.getString("siteArea"));
165                         wound.setSitePosition(woundJSON.getString("sitePosition"));
166                         wound.setSiteLocation(woundJSON.getString("siteLocation"));
167                         wound.setSiteSide(woundJSON.getString("siteSide"));
168                         wound.setWoundSize(woundJSON.getString("woundSize"));;
169                         wound.setActive(woundJSON.getBoolean("active"));
170                         String visitDate = woundJSON.getString("dateFound");
171                         wound.setDateFound(convert.String2Date2(visitDate));
172                         patient.getWound().add(wound);
173                         patientService.merge(patient);
174                         return Response.ok(patient).build();
175                     }
176             } else {
177                 ResponseBuilder builder = Response.status(Status.BAD_REQUEST);
```

```
178                     builder.type("application/json");
179                     builder.entity("<error>Patient Not Found</error>");
180                     return Response.status(Status.BAD_REQUEST).build();
181                 }
182             } catch (JSONException ex) {
183                 Logger.getLogger(PatientResourceImpl.class.getName()).log(Level.SEVERE, null, ex);
184                 return Response.status(Status.BAD_REQUEST).build();
185             }
186         }
187     }
```

Listing A.2: API implementation for the Patient Resource that produces JSON format only

```
1  public interface IdentificationDocumentTypeResource {
2      public abstract Collection<IdentificationDocumentType> getIdentificationDocumentType();
3      public abstract IdentificationDocumentType getIdentificationDocumentType(Long Id);
4      public abstract IdentificationDocumentType getIdentificationDocumentType (@Context Request request, RequestForm form) throws IdentificationDocumentTypeNotFoundException;
5      public abstract Response createIdentificationDocumentType (@Context Request request, IdentificationDocumentTypeForm form);
6      public abstract Response updateIdentificationDocumentType (@Context Request request, IdentificationDocumentType identificationDocumentType);
7  }
```

Listing A.3: API Interface code for Identification retrieval

```
1
2     @Path("/indentificationdocumenttypeservice")
```

```
3  @Produces({"application/xml","application/json"})
4  @Service("indentificationDocumentTypeResource")
5  public class IdentificationDocumentTypeResourceImpl ↩
       implements IdentificationDocumentTypeResource{
6
7       @Autowired
8       private IdentificationDocumentTypeService ↩
       identificationDocumentTypeService;
9
10      public IdentificationDocumentTypeService ↩
       getIdentificationDocumentTypeService() {
11          return identificationDocumentTypeService;
12      }
13
14      public void setIdentificationDocumentTypeService(↩
       IdentificationDocumentTypeService ↩
       identificationDocumentTypeService) {
15          this.identificationDocumentTypeService = ↩
       identificationDocumentTypeService;
16      }
17
18      @GET
19      @Path("/identificationdocumenttype")
20      @Override
21      public Collection<IdentificationDocumentType> ↩
       getIdentificationDocumentType() {
22          return identificationDocumentTypeService.findAll();
23      }
24
25      @GET
26      @Path("/identificationdocumenttype/{id}")
27      @Override
28      public IdentificationDocumentType ↩
       getIdentificationDocumentType(@PathParam("id") Long id) {
29          IdentificationDocumentType ↩
       identificationDocumentType = ↩
       identificationDocumentTypeService.find(id);
30          if (identificationDocumentType == null) {
```

```
31              ResponseBuilder builder = Response.status(Status.BAD_REQUEST);
32              builder.type("application.xml");
33              builder.entity("<error>Adress Not Found</error>");
34              throw new WebApplicationException(builder.build());
35          } else {
36              return identificationDocumentType;
37          }
38      }
39
40      @POST
41      @Path("identificationdocumenttype")
42      @Override
43      public IdentificationDocumentType getIdentificationDocumentType(@Context Request request, RequestForm form) {
44          try {
45              IdentificationDocumentType identificationDocumentType = (IdentificationDocumentType) identificationDocumentTypeService.getByPropertyName(form.getPropertyName(), form.getPropertyValue());
46              return identificationDocumentType;
47          } catch (IdentificationDocumentTypeNotFoundException ex) {
48              ResponseBuilder builder = Response.status(Status.BAD_REQUEST);
49              builder.type("application/xml");
50              builder.entity("<error>Invalid Code</error>");
51              Logger.getLogger(IdentificationDocumentTypeResourceImpl.class.getName()).log(Level.SEVERE, null, ex);
52              throw new WebApplicationException(builder.build());
53          }
54      }
55
```

```
56      @POST
57      @Path("identificationdocumenttypes")
58      @Override
59      public Response createIdentificationDocumentType(↩
       @Context Request request, IdentificationDocumentTypeForm ↩
       form) {
60          IdentificationDocumentType ↩
       identificationDocumentType = new  ↩
       IdentificationDocumentType();
61          identificationDocumentType.setType(form.getType());
62          identificationDocumentType.setDescription(form.↩
       getDescription());
63
64          try {
65              identificationDocumentTypeService.persist(↩
       identificationDocumentType);
66              ResponseBuilder builder = Response.created(new ↩
       URI("http://localhost:8084/hashmed/ws/↩
       identificationdocumenttypeservice/↩
       identificationdocumenttypes"));
67              return builder.build();
68          } catch (URISyntaxException e){
69                  throw new RuntimeException(e);
70          }
71      }
72
73      @PUT
74      @Path("identificationdocumenttype/update")
75      @Override
76      public Response updateIdentificationDocumentType(↩
       @Context Request request, IdentificationDocumentType ↩
       identificationDocumentType) {
77          if (identificationDocumentType == null){
78              return Response.status(Status.BAD_REQUEST).build↩
       ();
79          } else {
80              identificationDocumentTypeService.merge(↩
       identificationDocumentType);
```

```
81                return Response.ok(identificationDocumentType).↩
        build();
82              }
83         }
84   }
```

Listing A.4: Implementation code that produces both XML and JSON data format for the client

Bibliography

Afullo, T. J. (2000), 'Global information and africa: the telecommunications infrastructure for cyberspace', *Library management* **21**(4), 205–214.

Alarcon, R., Wilde, E. and Bellido, J. (2011), Hypermedia-driven restful service composition, *in* 'Service-Oriented Computing', Springer, pp. 111–120.

Altmanninger, K., Seidl, M. and Wimmer, M. (2009), 'A survey on model versioning approaches', *International Journal of Web Information Systems* **5**(3), 271–304.

Balter, R., Bernadat, J., Decouchant, D., Duda, A., Freyssinet, A., Krakowiak, S., Meysembourg, M., Van, P. L. D. H. N., Paire, E., Riveill, M. et al. (1991), 'Architecture and implementation of guide, an object-orienteddistributedsystem', *Computing* **4**, 31–67.

Banavar, G., Chandra, T., Strom, R. and Sturman, D. (1999), 'A case for message oriented middleware', *Distributed Computing* pp. 846–846.

Beck, K. (2003), *Test-driven development: by example*, Addison-Wesley Professional.

Beck, K. and Andres, C. (2004), *Extreme Programming Explained: Embrace Change (2nd Edition)*, Addison-Wesley Professional.

Berners-Lee, T., Fielding, R. and Masinter, L. (1998), 'Uniform resource identifiers (uri): generic syntax'.

Bernstein, P. (1996), 'Middleware: a model for distributed system services', *Communications of the ACM* **39**(2), 86–98.

Birman, K. (1997), 'Building secure and reliable network applications', *Worldwide Computing and Its Applications* pp. 15–28.

Brown, A., Johnston, S. and Kelly, K. (2002), 'Using service-oriented architecture and component-based development to build web service applications', *Rational Software Corporation* .

CEN, T. (2002*a*), '251', *N02-015, prEN* **1064**.

CEN, T. (2002*b*), '251 (European Standardization of Health Informatics) ENV 13606, Electronic Health Record Communication'.

Chahal, K. K. and Singh, H. (2009), 'Metrics to study symptoms of bad software designs', *SIGSOFT Softw. Eng. Notes* **34**(1), 1–4.
URL: *http://doi.acm.org/10.1145/1457516.1457522*

Chen, I. Y. and Tsai, C. H. (2008), 'Pervasive digital monitoring and transmission of precare patient biostatics with an osgi, mom and soa based remote health care system', pp. 704–709.

Chute, C. (2000), 'Clinical Classification and Terminology', *Journal of the American Medical Informatics Association* **7**(3), 298.

Chute, C., Cohn, S., Campbell, K., Oliver, D. and Campbell, J. (1996), 'The content coverage of clinical classifications', *Journal of the American Medical Informatics Association* **3**(3), 224.

Collins-Sussman, B., Fitzpatrick, B. and Pilato, M. (2007), *Version control with subversion*, O'Reilly Media.

Conradi, R. and Westfechtel, B. (1998), 'Version models for software configuration management', *ACM Computing Surveys (CSUR)* **30**(2), 232–282.

Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N. and Weerawarana, S. (2002), 'Unraveling the Web services web: an introduction to SOAP, WSDL, andUDDI', *IEEE Internet computing* **6**(2), 86–93.

Curry, E. (2004), 'Message-Oriented Middleware', *Middleware for communications* p. 1.

DeRenzi, B., Anokwa, Y., Parikh, T. and Borriello, G. (2007), Reliable data collection in highly disconnected environments using mobile phones, *in* 'Proceedings of the 2007 workshop on Networked systems for developing regions', ACM, p. 4.

Detmer, D. E. (2003), 'Building the national health information infrastructure for personal health, health care services, public health, and research', *BMC Medical Informatics and Decision Making* **3**(1), 1472–6947.

Dhama, H. (1995), 'Quantitative models of cohesion and coupling in software', *Journal of Systems and Software* **29**(1), 65–74.

Dogac, A., Namli, T., Okcan, A., Laleci, G., Kabak, Y. and Eichelberg, M. (2006), 'Key issues of technical interoperability solutions in ehealth', *Proceedings of eHealth 2006 High Level Conference Exhibition and Associated Events,, Malaga, Spain, May* .

Dolin, R., Alschuler, L., Boyer, S., Beebe, C., Behlen, F., Biron, P. and Shabo Shvo, A. (2006), 'HL7 clinical document architecture, release 2', *Journal of the American Medical Informatics Association* **13**(1), 30.

Drucker, P. F. et al. (1988), 'The coming of the new organization', **66**(1), 45–53.

Eichelberg, M., Aden, T., Riesmeier, J., Dogac, A. and Laleci, G. B. (2005), 'A survey and analysis of electronic healthcare record standards', *ACM Computing Surveys (CSUR)* **37**(4), 277–315.

Evans, E. (2004), *Domain-driven design: Tackling complexity in the heart of software*, Longman.

Fall, K. (2004), 'Messaging in difficult environments', *Intel Research Berkeley, IRB-TR-04-019* .

Fielding, R. (2000), Architectural styles and the design of network-based software architectures, PhD thesis, Citeseer.

Fowler, M. (2010), 'Richardson maturity model: steps toward the glory of rest', *Online at http://martinfowler. com/articles/richardsonMaturityModel. html* .

Fraser, H. S. F., Biondich, P., Moodley, D., Choi, S., Mamlin, B. W. and Szolovits, P. (2005), 'Implementing electronic medical record systems in developing countries', *Informatics in Primary Care* **13**(2), 83–96.

Garde, S., Hovenga, E., Buck, J. and Knaup, P. (2007), 'Expressing clinical data sets with openEHR archetypes: A solid basis for ubiquitous computing', *International journal of medical informatics* **76**, S334–S341.

Garg, V. (2002), *Elements of distributed computing*, Wiley-IEEE Press.

Geihs, K. (2001), 'Middleware challenges ahead', *IEEE computer* **34**(6), 24–31.

Ghezzi, C., Jazayeri, M. and Mandrioli, D. (2002), *Fundamentals of software engineering*, Prentice Hall PTR.

Gibbons, P., Arzt, N., Burke-Beebe, S., Chute, C., Dickinson, G., Flewelling, T. et al. (2007), 'Coming to Terms: scoping interoperability for healthcare', *Health Level* **7**.

Goetsch, D. and Davis, S. (1998), *Understanding and implementing ISO 9000 and ISO Standards*, Prentice Hall.

Goodhue, D. L., Wybo, M. D. and Kirsch, L. J. (1992), 'The impact of data integration on the costs and benefits of information systems', *MIS Q.* **16**(3), 293–311.

Govindaraju, M., Slominski, A., Chiu, K., Liu, P., Van Engelen, R. and Lewis, M. (2004), Toward characterizing the performance of SOAP toolkits, *in* 'Fifth IEEE/ACM International Workshop on Grid Computing, 2004. Proceedings', pp. 365–372.

Green, T. J., Karvounarakis, G., Taylor, N. E., Biton, O., Ives, Z. G. and Tannen, V. (2007), Orchestra: facilitating collaborative data sharing, *in* 'Proceedings of the 2007 ACM SIGMOD international conference on Management of data', ACM, pp. 1131–1133.

Greenfield, J. and Short, K. (2003), Software factories: assembling applications with patterns, models, frameworks and tools, *in* 'Conference on Object Oriented Programming Systems Languages and Applications', ACM New York, NY, USA, pp. 16–27.

Grimson, J., Grimson, W. and Hasselbring, W. (2000), 'The SI challenge in health care', *Communications of the ACM* **43**(6), 48–55.

Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J., Nielsen, H., Karmarkar, A. and Lafon, Y. (2003), 'SOAP version 1.2 part 1: Messaging framework'.

Guo, J., Takada, A., Tanaka, K., Sato, J., Suzuki, M., Suzuki, T., Nakashima, Y., Araki, K. and Yoshihara, H. (2004), 'The development of MML (Medical Markup Language) version 3.0 as a medical document exchange format for HL7 messages', *Journal of Medical Systems* **28**(6), 523–533.

Guo, S., Kern, R. J., Lee, B. and Ostler, P. A. (2008), 'Optimistic locking in online and offline environments'. US Patent App. 12/099,620.

Gupta, A. (1989), *Integration of information systems: Bridging heterogeneous databases*, IEEE press.

Halevy, A., Ashish, N., Bitton, D., Carey, M., Draper, D., Pollock, J., Rosenthal, A. and Sikka, V. (2005), Enterprise information integration: successes, challenges and controversies, *in* 'Proceedings of the 2005 ACM SIGMOD international conference on Management of data', ACM, p. 787.

Hashimi, S. (2003), 'Service-oriented architecture explained', *ONDot Net. com, August* **18**.

Hegering, H., Abeck, S. and Neumair, B. (1999), *Integrated management of networked systems: concepts, architectures, and their operational application*, Morgan Kaufmann Pub.

Hempel, C. G. (1966), *Philosophy of natural science*, Prentice-Hall Englewood Cliffs, NJ.

Hernández, A. G. and García, M. N. M. (2010), A formal definition of restful semantic web services, *in* 'WS-REST '10: Proceedings of the First International Workshop on RESTful Design', ACM, New York, NY, USA, pp. 39–45.

Hersh, W. (2004), 'Health Care Information Technology: Progress and Barriers', *Jama* **292**(18), 2273.

Hevner, A., March, S., Park, J. and Ram, S. (2004), 'Design science in information systems research', *Mis Quarterly* **28**(1), 75–105.

Hodges, S., Mijumbi, C., Okello, M., McCormick, B., Walker, I. and Wilson, I. (2007), 'Anaesthesia services in developing countries: defining the problems', *Anaesthesia* **62**(1), 4–11.

Hohpe, G. and Woolf, B. (2003), *Enterprise integration patterns: Designing, building, and deploying messaging solutions*, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

Hovenga, E. J. S. (2004), 'Globalisation of health and medical informatics education—what are the issues?', *International Journal of Medical Informatics* **73**(2), 101–109.

Hovenga, E., Kidd, M. and Cesnik, B. (1996), 'Standards in health informatics', *Health informatics: An Overview. Melbourne: Churchill Livingston* pp. 41–46.

Iakovidis, I. (1998), 'Towards personal health record: current situation, obstacles and trends in implementation of electronic healthcare record in Europe', *International journal of medical informatics* **52**(1-3), 105–115.

Jensen, M. (2002), 'Ict in africa: A status report', *The global information technology report* **2003**.

Kalra, D. (2004), 'CEN prEN 13606, draft standard for Electronic Health Record Communication, and its introduction to ISO TC/215. Document CEN', *TC* **251**.

Kalra, D., Beale, T. and Heard, S. (2005), 'The openEHR foundation', *Studies in Health Technology and Informatics* **115**, 153–173.

Kaplan, A. (1964), *The conduct of inquiry: Methodology for behavioral science*, Transaction Pub.

Kendall, S. C., Waldo, J., Wollrath, A. and Wyant, G. (1994), A note on distributed computing, Technical report, Mountain View, CA, USA.

Kenny, C. (2000), 'Expanding Internet access to the rural poor in Africa [*] The views expressed in this paper are those of the author, and do not reflect those of the World Bank or its executive directors. This paper is based partly on results presented in The Economic', *Information Technology for Development* **9**(1), 25–31.

Korpela, M. (2013), Social, constructive and natural sciences in relation to basic ontological and epistemological assumptions, *in* '4th biannual ISD4D seminar Maputo, Mozambique, 11-13 April 2013', INDEHELA, p. 6.

Korpela, M., Mykkanen, J., Porrasmaa, J., Sipila, M. and Unit, H. (2005), 'Software architectures for digital healthcare: Experiences and views'.

Leiner, B. M., Cerf, V. G., Clark, D. D., Kahn, R. E., Kleinrock, L., Lynch, D. C., Postel, J., Roberts, L. G. and Wolff, S. S. (1997), 'The past and future history of the internet', *Communications of the ACM* **40**(2), 102–108.

Linthicum, D. (2000), *Enterprise application integration*, Addison-Wesley Longman Ltd. Essex, UK, UK.

Lublinsky, B. and Tyomkin, D. (2003), 'Dissecting service-oriented architectures', *Business Integration Journal* **10**, 52–58.

Mandl, K. D., Szolovits, P. and Kohane, I. S. (2001), 'Public standards and patients' control: how to keep electronic medical records accessible but private', *BMJ: British Medical Journal* **322**(7281), 283.

March, S. T. and Smith, G. F. (1995), 'Design and natural science research on information technology', *Decision support systems* **15**(4), 251–266.

Martin, R. C. (1996), 'The open-closed principle', *More C++ gems* pp. 97–112.

Martin, R. C. (2003), *Agile software development: principles, patterns, and practices*, Prentice Hall PTR.

Mattsson, M., Bosch, J. and Fayad, M. E. (1999), 'Framework integration problems, causes, solutions', *Communications of the ACM* **42**(10), 80–87.

McGovern, J., Ambler, S., Stevens, M., Linn, J., Jo, E. and Sharan, V. (2003), *The practical guide to Enterprise Architecture*, Prentice Hall PTR Upper Saddle River, NJ, USA.

Mead, C. N. et al. (2006), 'Data interchange standards in healthcare it-computable semantic interoperability: Now possible but still difficult. do we really need a better mousetrap?', *Journal of Healthcare Information Management* **20**(1), 71.

Meijler, T., Kruithof, G. and van Beest, N. (2006), 'Top down versus bottom up in service-oriented integration: an MDA-based solution for minimizing technology coupling', *Service-Oriented Computing–ICSOC 2006* pp. 484–489.

Menge, F. (2007), Enterprise service bus, *in* 'Free and open source software conference', Vol. 2, pp. 1–6.

Mildenberger, P., Eichelberg, M. and Martin, E. (2002), 'Introduction to the DICOM standard.', *European radiology* **12**(4), 920.

Mori, K., Ihara, H., Kawano, K., Koizumi, M., Orimo, M., Nakai, K., Nakanishi, H. and Suzuki, Y. (1986), Autonomous decentralized software structure and its application, *in*

'ACM '86: Proceedings of 1986 ACM Fall joint computer conference', IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 1056–1063.

Mukherjee, P., Leng, C., Terpstra, W. W. and Schurr, A. (2008), Peer-to-peer based version control, *in* 'Parallel and Distributed Systems, 2008. ICPADS'08. 14th IEEE International Conference on', IEEE, pp. 829–834.

Mutula, S. M. (2008), 'Digital divide and economic development: case study of sub-saharan africa', *Electronic Library, The* **26**(4), 468–489.

Mutula, S. M. and P, V. B. (2007), 'Ict skills readiness for the emerging global digital economy among small businesses in developing countries: Case study of botswana', *Library Hi Tech* **25**(2), 231–245.

Narten, T. (2008), 'Guidelines for writing an iana considerations section in rfcs'.

Natis, Y. (2003), 'Service-oriented architecture scenario', *Gartner Research Note AV-19-6751* .

Ng, A., Chen, S. and Greenfield, P. (2004), An evaluation of contemporary commercial SOAP implementations, *in* 'Proceedings of the 5th Australasian Workshop on Software and System Architectures (AWSA 2004', Citeseer, pp. 64–71.

Nixon, R. (2007), 'Africa, Offline: Waiting for the web', *New York Times* **22**.

Nordberg III, M. E. (2001), Aspect-oriented dependency inversion, *in* 'OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems', Citeseer.

OASIS, S. (n.d.), 'Reference Model Technical Committee, OASIS Reference Model for Service Oriented Architecture 1.0'.

Oates, B. J. (2005), *Researching information systems and computing*, Sage Publications Limited.

Papazoglou, M. P. and Georgakopoulos, D. (2003), 'Service-oriented computing', *Communications of the ACM* **46**(10), 25–28.

Pautasso, C., Zimmermann, O. and Leymann, F. (2008), Restful web services vs. big'web services: making the right architectural decision, *in* 'Proceeding of the 17th international conference on World Wide Web', ACM, pp. 805–814.

Primer, A. (2006), 'SOAP-Based Web Services', *Beginning Web Development with Perl* pp. 137–152.

Quinn, J. (1998), 'An HL7 (Health Level Seven) overview.', *Journal of AHIMA/American Health Information Management Association* **70**(7), 32.

Ruh, W. A., Brown, W. J. and Maginnis, F. X. (2000), *Enterprise Application Integration: A Wiley Tech Brief*, John Wiley & Sons, Inc., New York, NY, USA.

Schmidt, D. (1994), 'A Domain Analysis of Network Daemon Design Dimensions', *C++ Report* **6**(3).

Schmidt, D. and Vinoski, S. (1997), 'Object interconnections. Object adapters: concepts and terminology', *SIGS C++ Report* **11**.

Schmidt, M., Hutchison, B., Lambros, P. and Phippen, R. (2005), 'The enterprise service bus: Making service-oriented architecture real', *IBM Systems Journal* **44**(4), 781–797.

ServiceMix, A. (2007), 'Apache servicemix 3. x users guide', *Apache ServiceMix community. URL http://incubator. apache. org/servicemix/users-guide. html.(Cited on pages 72, 73 and 149.)* .

Sheltzer, A. and Popek, G. (1986), 'Internet Locus: Extending transparency to an internet environment.', *IEEE Transactions on Software Engineering* **12**(11), 1067–1075.

Siegel, E. and Channin, D. (2001), 'Integrating the healthcare enterprise: a primer', *Radiographics* **21**(5), 1339.

Simon, H. A. (1996), *The sciences of the artificial*, MIT press.

Sluis, D., Lee, K. and Mankovich, N. (2002), 'DICOM SR-integrating structured data into clinical information systems', *Medicamundi* **46**(2), 31–36.

Snell, J., Tidwell, D. and Kulchenko, P. (2002), *Programming Web services with SOAP*, O'Reilly & Associates, Inc., Sebastopol, CA, USA.

Sullivan, F. (2001), 'What is health informatics?', *Journal of Health Services Research & Policy* **6**(4), 251.

Tan, J. and Payton, F. (2009), *Adaptive Health Management Information Systems: Concepts, Cases, and Practical Applications*, Jones & Bartlett Pub.

Thiran, P., Hainaut, J.-L., Houben, G.-J. and Benslimane, D. (2006), 'Wrapper-based evolution of legacy information systems', *ACM Trans. Softw. Eng. Methodol.* **15**(4), 329–359.

Tsai, W. T. (2005), 'Service-oriented system engineering: A new paradigm', pp. 3–8.

Twidale, M. B. and Floyd, I. (2008), Infrastructures from the bottom-up and the top-down: can they meet in the middle?, *in* 'PDC '08: Proceedings of the Tenth Anniversary Conference on Participatory Design 2008', Indiana University, Indianapolis, IN, USA, pp. 238–241.

Wagh, K. and Thool, R. (2012), 'A comparative study of soap vs rest web services provisioning techniques for mobile host', *Journal of Information Engineering and Applications* **2**(5), 12–16.

Walker, J., Pan, E., Johnston, D., Adler-Milstein, J., Bates, D. and Middleton, B. (2005), 'The Value Of Health Care Information Exchange And Interoperability', *Health Affairs* .

Wallsten, S. (2001), 'An econometric analysis of telecom competition, privatization, and regulation in Africa and Latin America', *Journal of Industrial Economics* pp. 1–19.

West, J. (2003), 'How open is open enough?:: Melding proprietary and open source platform strategies', *Research Policy* **32**(7), 1259–1285.

Wilson, E. V. and Tulu, B. (2010), 'The rise of a health-it academic focus', *Commun. ACM* **53**(5), 147–150.

Woodley, T. and Gagnon, S. (2005), 'BPM and SOA: Synergies and Challenges', *Web Information Systems Engineering–WISE 2005* pp. 679–688.

Yu, Q., Liu, X., Bouguettaya, A. and Medjahed, B. (2008), 'Deploying and managing web services: issues, solutions, and directions', *The VLDB Journal* **17**(3), 537–572.

Zur Muehlen, M., Nickerson, J. and Swenson, K. (2005), 'Developing web services choreography standards–the case of REST vs. SOAP', *Decision Support Systems* **40**(1), 9–29.