2006

A Framework For Improving Tractability In Software Development

Sambit Patnaik
University of Alabama at Birmingham

A FRAMEWORK FOR IMPROVING TRACTABILITY IN SOFTWARE DEVELOPMENT

by

SAMBIT PATNAIK

MURAT M. TANIK, COMMITTEE CHAIR

GARY J. GRIMES

MURAT N. TANJU

A THESIS

Submitted to the graduate faculty of The University of Alabama at Birmingham,
in partial fulfillment of the requirements for the degree of
Master of Science

BIRMINGHAM, ALABAMA

2006

A FRAMEWORK FOR IMPROVING TRACTABILTY IN SOFTWARE DEVELOPMENT

SAMBIT PATNAIK

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

ABSTRACT

Present software development methodologies suffer from a common disadvantage that they do not allow seamlessly mapping the final implementations to the originating requirements. Most of the advances made in this area have been in the form of policy rather than processes and framework which support that kind of tractability. New software environments constituting component-based software promise to change the present picture.

Web services and semantic web are upcoming technologies. Their popular use in the software industry is inevitable, dependent only upon how fast they mature into reliable technologies. Web services evolved from component software technology which is based upon the concept of software reusability. Reusability of functionalities has been in existence since object oriented programming came into being. Semantic web technology on the other hand is an aggregation of various technologies and is a concept which is being implemented in different ways by different research groups. The common ground of most of these groups is representative data formats like XML and rule engines which would be able to act automatically on the data.

Since the aforementioned technologies are new, they haven't been used in collaboration to develop a software environment. The objective of my research is to build such a framework which makes uses of web services and semantic web and to deal with

some of the problems associated with software development most markedly the intractability of requirements.

DEDICATION

I dedicate this thesis to Dr. Tanik for his insight into web services which motivated me to think in those directions. I am thankful to my brother for continuously educating me about various J2EE technologies.

TABLE OF CONTENTS

Page

APPENDIX

LIST OF FIGURES

Figure Page

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
API	Application Programming Interface
CA	Certificate Authority
CORBA	Common Object Request Broker Architecture
CRL	Certificate Revocation List
DAML	DARPA Agent Markup Language
DIME	Direct Internet Message Encapsulation
DOM	Document Object Model
DTD	Document Type Definition
EJB	Enterprise Java Beans
FIPA	Foundation for Intelligent Physical Agents
HTML	Hyper Text Markup Language
HTTP	Hyper-Text Transfer Protocol
J2EE	Java 2 Enterprise Edition
JAXP	Java API for XML Processing
JDBC	Java Database Connectivity
JMS	Java Message Service
JMX	Java Management Extensions
JSP	Java Server Pages

LDAP	Lightweight Directory Access Protocol
MIME	Multipurpose Internet Mail Extensions
OASIS	Organization for the Advancement of Structured Information Standards
RAD	Rapid Application Design
RDBMS	Relational Database Management Service
RDF	Resource Description Framework
RDFS	RDF Schema
RDQL	RDF Query Language
RUP	Rational Unified Process
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SSO	Single Sign On
UDDI	Universal Description, Discovery, and Integration
UI	User Interface
URI	Uniform Resource Identifier
W3C	World-Wide Web Consortium
WMI	Windows Management Instrumentation
WSDL	Web Services Description Language
XDB	XML Database
XML	Extensible Markup Language

CHAPTER 1

INTRODUCTION: THE AIM OF THE RESEARCH

The primary aim of the research is to examine some of the problems faced in the development of commercial software, in particular, the problems faced by the technical team in mapping the user requirements to the software architecture, the architecture to low level design, and the design to the final implementation in the form of software modules.

In present day software development practice, there are various aspects of user requirements that are unaccounted for as one moves from the requirement stage to the end of the development phase. Each of these phases is loosely coupled to the other; hence there is considerable effort spent on connecting the initial requirements with the final implementations in the intermediate phases. During each of the intermediate phases, the development group spends a considerable amount of effort in analyzing the work product that was created as a result of that particular phase and in verifying whether the product satisfies the specifications that were laid out in the previous phase. For example, during the development phase, the code reviews and unit test cases verify the correctness of the code with respect to the previous phase's specification, i.e., the design solution. Similarly, in the design phase, the design team analyzes how well their design conforms to the architecture laid out by the architect and if any aspect has been overlooked. The system architects in the architecture phase attempt to capture each functional specification in their model and examine in detail all the functional specifications so that

none of it is left out or misinterpreted. This kind of verification, which is present in almost all of the phases of software development and comprises an entire software development phase by itself (system testing and user testing, which on an average account for approximately 30% of the total software development time), uses up a major portion of the total development effort. Ready-to-use components would be a big factor in cutting down such effort because of their reusability, hence the popularity of component architecture and web services since they were developed.

The goal is to focus on various technical solutions that will result in greater efficiency in software development; use of component-based technologies or automation technologies will be the foremost approaches. The primary focus of this research is a case study of a framework that can achieve effectiveness in software development by dealing effectively with some of the major problems. The problems are described in greater detail in Chapter 2. In the process of designing such a framework, considerable time will be spent in researching the various technical options available. Web services and semantic web technologies have been explored in detail and are found to be appropriate for building such a framework. The basic understanding of each of these technologies is explained in the later chapters of this document. These two technologies are presently evolving and might end up in being of much greater use in solving the problems that are being addressed. The importance of web services in software development is widely recognized already, and various research organizations are focusing on its use in this scenario [1]. Some of the problems that are being faced by software architects are also described, along with a proposed framework for solving them. Software architecture faces problems primarily because of the limitations of the available tools for architecture.

Software architecture would benefit greatly from more advanced and comprehensive tools or from frameworks that make better use of presently available tools. Making use of web services and the semantic web in a typical software development scenario would also help in developing a better understanding of these technologies.

CHAPTER 2

PROBLEMS IN THE CURRENT SOFTWARE DEVELOPMENT SCENARIO

Software engineering, as practiced in all major software development organizations, follow certain norms as regards models or methodology. Of the many methodologies in existence, there are a few very popular models that have been in existence for a good period of time. The following section briefly introduces each of those methodologies.

2.1 Software Development Methodologies

2.1.1 Waterfall Model

In the waterfall model, the phases of software engineering are in sequence and one phase must be completed before the succeeding one can start. The development process starts with the requirements specification phase and ends with the maintenance phase. When the requirement phase is completed, the development team moves ahead with the architecture and then the design phases. After the design phase is completed, the developers begin the application coding and, at the end of the development phase, various implementations are integrated to create the complete product. Thus the waterfall model maintains that one should move to a phase only when it's earlier phase is completed and perfected. The various phases of the waterfall model in sequence are as follows:

- Requirements specifications,
- Design,

- Construction (involves design and coding),
- Integration,
- Testing,
- Installation, and
- Maintenance.

Major disadvantages of the waterfall model from the stand point of this research are as follows: this model mandates that the requirements should be perfectly laid out at the end of the requirements specification phase and that the various implemented modules at the end of construction must be integrated. These and other disadvantages will be discussed later in this chapter. The waterfall model has been the most popular of all of the software development methodologies and the problems associated with this model are also present in most of the software projects that use the pure form of this methodology.

2.1.2 Iterative Models

The iterative models are based on the idea that software can be developed incrementally, so that the knowledge gained in one cycle can be applied to succeeding cycles, and the errors can be minimized by the time the full product is implemented. The basic idea behind iterative enhancement is to develop a software system incrementally, allowing the developer to take advantage of what was being learned during the development of earlier, incremental, deliverable versions of the system. At the end of every iteration, design modifications are made and new functional capabilities added. The process can be divided into three aspects: the initialization, which creates a base version of the project in which the most basic features of the system are implemented; the

iteration step which involves the redesign and implementation of the task; and project control list to guide the iteration process. The project control list includes such items as new features to be implemented and areas of redesign of the existing solution. The control list is constantly being revised as a result of the analysis phase [2]. Rational Unified Process (RUP) is an iterative model of software development and is widely used in the industry. RUP is an advanced version of the unified process for software development. The RUP deals with most of the problems that were discussed in the first chapter using a set of software development principles [4]. These principles are as follows:

- Develop software iteratively,
- Manage requirements,
- Use component-based architecture,
- Model software visually,
- Verify software quality, and
- Control changes to software.

Using component-based architecture of software development forms an important principle of the RUP. Requirement management is defined by the following set of guidelines:

- The correct requirements generate the correct product; the customer's needs are met and
- Necessary features will be included, reducing post-development costs.

2.1.3 Rapid Application Development

Rapid application development (RAD) is a development methodology that takes advantage of automated tools and techniques to restructure the process of building information systems. RAD operates exclusive of software practices that require a minimum of labor intensive design and coding practices that are related to individual performance. It relies instead on automated design and coding, which is an inherently more stable process [3]. In addition to being more stable, RAD, as depicted in Figure 1, is a more capable process. It is much faster and less error-prone than hand coding. In addition to reduced development time and effort optimization, RAD also decreases the effort expended on the maintenance of existing systems, which typically have minimum documentation and legacy development using legacy technologies.

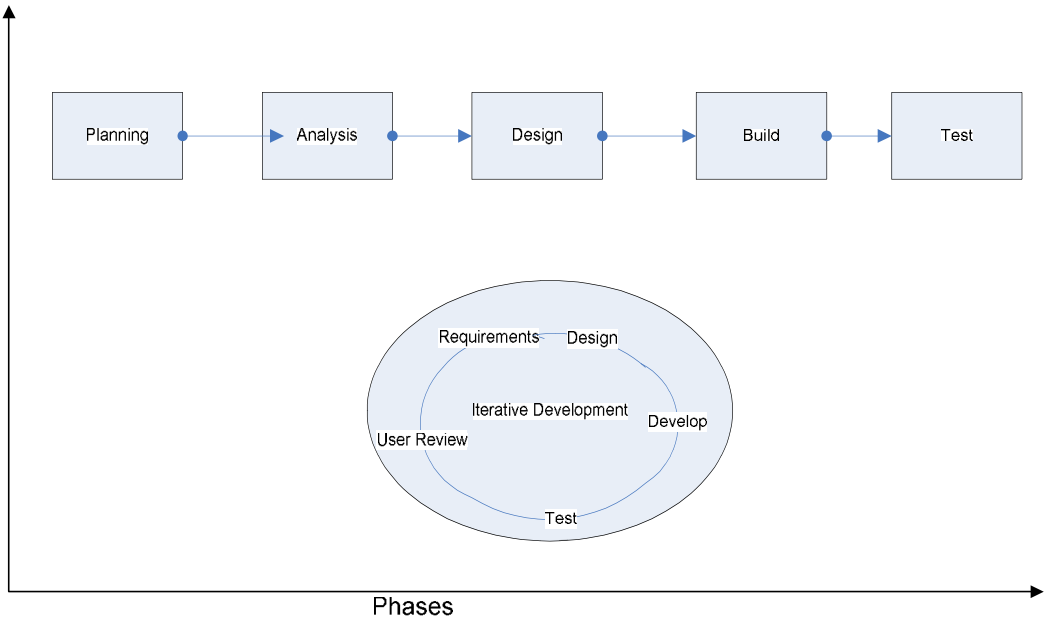

Figure 1. Traditional software development vs. RAD

RAD depends on effective implementation of four essential factors: the development methodology, the development team, the software management group, and the tools used for development. If any one of these factors is inadequate, the development will not progress at the expected pace. Development life cycles, which weave these ingredients together as effectively as possible, are of the utmost importance.

Rapid prototyping is another form of software development methodology that focuses on requirements management. A prototype is defined as an initial version of the desired system. A prototype involves partial implementation of the total set of requirements; the aim being analysis to arrive at the final product rather than using the prototype system as a production system. Since the construction activity to build a prototype is rapid, the practice itself is called rapid prototyping. Rapid prototyping is an effective technique for clarifying requirements and eliminating the large amount of effort currently wasted on developing software to meet incorrect or inappropriate requirements in traditional software life cycles. A lack of agreement on the requirements as specified by the customer and as analyzed by the designer is a cause of inconsistencies between the delivered system and customer expectations, leading to expensive rebuilding. This problem is especially acute for large systems and systems with real-time constraints because the requirements for such systems are complicated and difficult to understand. The requirements are firmed up iteratively in a rapid prototyping approach through the examination of executable prototypes as well as by negotiations between customer and designer. The designer constructs a prototype based on the requirements and examines the execution of the prototype together with the customer. The requirements are adjusted

based on feedback from the customer, and the prototype is modified accordingly until both the customer and the designer agree on the requirements [8].

2.1.4 Cleanroom

Cleanroom development methodology originated from the semi-conductor manufacturing industry and emphasizes on defect prevention rather than the detection and removal of defects from the software products. Most of the effort in Cleanroom methodology therefore involves producing defect-free software before it goes to testing. Cleanroom methodology defines the software system or component by a total function called a black box function. The domain is the set of all possible sequences of input stimuli, including illegal sequences, and the range is the set of possible system responses. The sequence-based specification method is used to derive the initial black box function from an informal specification; it is the vital bridge between formal and informal worlds. The starting point is the informal, natural language requirements specification, written according to current practice in domain-specific terms. The original terms and domain concepts are used together with the traceability; this enables the critical project stakeholders to establish by inspection that the formal specification specifies the same system as the informal specification. This addresses the problem of different starting points and the need for traceability between the formal specification and the customer's requirements specifications [5].

2.2 Problems in Software Development

There are various problems associated with current software development techniques. The pure form of the waterfall model, which is the oldest software development method, is associated with many of the problems listed below. Many of the aforementioned development technologies attempt to solve some of these problems, but none of them except RUP provide a comprehensive solution to each. The following is a representative list of problems associated with software development:

- Ambiguous and imprecise communication,
- Brittle architecture (architecture that does not work properly under stress),
- Overwhelming complexity,
- Undetected inconsistencies in requirements, designs, and implementations,
- Insufficient testing, and
- Subjective assessment of project status.

A problem that affects almost every phase is imprecise and erroneous communication, which starts during the requirements specification. This happens because of a lack of guiding principles for directing communications between the involved parties. Imprecise communication at the requirement phase between the customer and software managers may be one of the biggest reasons for the introduction of errors into the software product even before it is built. A proper framework for standardizing communication and capturing requirements should be established.

The next point, brittle architecture, is a result of the inability of the architecture team to imagine every aspect of the software product being developed. The architecture generally fails when the system size increases and the initial architecture envisioned for

the system does not scale accordingly. For a tightly formed architecture, which would hold true in all scenarios, the architecture team should brainstorm all use cases and use correct tools or other available methodologies to estimate the various use cases and performance requirements for the product. Reusing tried and tested sub-systems (equivalent to component architecture) is a possible way of avoiding architecture problems later in the development cycle.

Complexity in most applications happens due to absence of a clean separation of functionalities and entangled code blocks. This results in difficulties in the analysis of existing code. This becomes harder when the original development team is no longer involved with application maintenance. The different platforms, languages, and physical locations of the servers, which are almost always involved in any large-scale enterprise application, can be a nightmare for any software team. Code complexities affect the total effort in building and maintaining a software system, hence the total investment made in it. Complexity can be avoided by using a reliable, well-tested development methodology and by making use of reliable architecture that focuses on avoiding complexity in the code.

In the absence of standards, the various aspects of software development, including the architecture, the requirements gathering, and the implementation processes, can be inconsistent and diverge from any optimized process. These inconsistencies may result in the requirements being different from the design, and the code being different from the design. This results in the intractability of the requirements to implementation. The most problematic result of such inconsistency would be that some requirement might

be ignored and the final product might end up not satisfying the requirement. This is the worst problem that can be faced by a software development team.

CHAPTER 3

A BACKGROUND OF SOA: WEB SERVICES

Service-oriented architecture (SOA) is a framework that is rapidly gaining popularity. The concept was defined by Sun Microsystems in the late 1990's to describe Jini, an environment for dynamic discovery and use of services over a network. Web services have taken the concept of services introduced by Jini technology and implemented it as services delivered over the web using such technologies as Extensible Markup Language (XML), Web Services Description Language (WSDL), Simple Object Access Protocol (SOAP), and Universal Description, Discovery, and Integration(UDDI). SOA is emerging as the premier integration and architecture framework in today's complex and heterogeneous computing environment. Legacy systems were based on proprietary set of application programming interfaces and required a high degree of coordination between the groups that built them. The possibility of inter-operability was remote for such systems. SOA would help organizations streamline their business processes to a much greater extent and increase their efficiency greatly. This would also allow companies to adapt to changing needs and competition by enabling software as a service concept. Presently many companies, including Google and eBay, have such practices in place. The way they use their services is by opening them up for use by other companies; eBay has opened up its auction service and Google its search service for use by other companies. The goal of eBay is to drive developers to make money around the eBay platform. Through the new Application Programming Interface (API), developers

can build custom applications that link to the online auction site and allow applications to submit items for sale. Such applications are typically aimed at sellers, since buyers must still head to ebay.com to bid on items. This type of strategy, however, will increase the customer base for eBay.

SOA and web services are two different things, but web services are the preferred standards-based way to realize SOA. This chapter provides an overview of SOA and the role of web services in realizing it.

SOA is an architectural style for building software applications that use services available in a network, such as the web. It promotes loose coupling between software components so that they can be reused. Applications in SOA are built based on services. A service is an implementation of a well-defined business functionality, and such services can then be consumed by clients in different applications or business processes.

SOA allows for the reuse of existing assets where new services can be created from an existing IT infrastructure of systems. In other words, it enables businesses to leverage existing investments by allowing them to reuse existing applications, and promises interoperability between heterogeneous applications and technologies. SOA provides a level of flexibility that wasn't possible before in the sense that:

- Services are self-contained and loosely coupled,
- Services can be dynamically discovered, and
- Composite services can be built from aggregates of other services.

Services are software components with well-defined interfaces that are implementation-independent. An important aspect of SOA is the separation of the service interface and implementation. Such services are consumed by clients that are not concerned with how

these services will execute their requests. SOA uses the find-bind-execute paradigm, as shown in Figure 2. In this paradigm, service providers register their service in a public registry, which is used by consumers to find services that match certain criteria. If the registry has such a service, it provides the consumer with a contract and an endpoint address for that service [6]. SOAP is a simple XML based protocol to let applications exchange information over hyper-text transfer protocol (HTTP). SOAP is used for exchanging data while accessing web services.

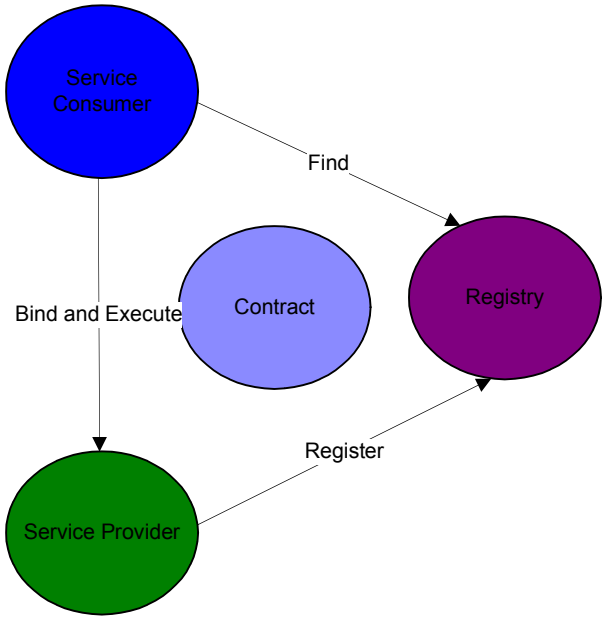

Figure 2. The find-bind-execute paradigm

3.1 Web Services Technologies

3.1.1 SOAP Messages

A SOAP message is an ordinary XML document, all the rules that are applicable to a regular XML document are also valid for SOAP. The structure of SOAP can be described as follows:

- SOAP Envelope: This element is used at the beginning of the SOAP message and is used to mark the XML document as a SOAP message
- SOAP Header: This element is used to specify the header information of the SOAP message.
- SOAP Body: This element contains the call and response information of the SOAP message.
- SOAP Fault: This element is optional and provides information about errors that occurred while processing the SOAP message.

The default namespace for the SOAP envelop for the above elements is defined in *http://www.w3.org/2001/12/soap-envelope*. The default namespace for SOAP encoding and data types is *http://www.w3.org/2001/12/soap*.

3.1.2 UDDI

UDDI is like any other business registry, with the exception that it uses XML for the exchange of information. UDDI allows a business entity to register itself on the internet. Once a business registers itself in UDDI, it can be found by any calling web service and can also be invoked by the web service. UDDI is platform-independent. UDDI is an open industry initiative sponsored by the Organization for the Advancement of Structured Information Standards (OASIS). In UDDI, a business entity publishes its service listings and defines how the services interact over the internet. UDDI is one of the core web service standards. The calling service finds all the information about the WSDL file of the called web service from UDDI. UDDI provides information such as protocol bindings and message formats about a registered web service, information that is required

by any calling web service. All of this information about the web service is listed in the directory of UDDI.

3.1.3 WSDL

A web service can be defined as a self-describing, self-contained, and modular unit of application logic that provides some functionality to other applications through a network connection. Applications access web services via ubiquitous web protocols and data formats, such as HTTP and XML, with no need to worry about how each web service is implemented. By providing a standards-based framework for exchanging information dynamically on demand between applications, web services show promise in addressing the information integration needs of an enterprise application.

In order to describe a web service, the features of the services should be listed in a file. The contents of the file would relate to the features of a web service; i.e., the input for the requested service, the format or data type, and the output. This file, which is called WSDL, would give an insight or a description of a particular web service. WSDL is an industry standard used for web service description, discovery, and invocation. The application that tries to use the web service first tries to read and interpret the WSDL. WSDL is a simple XML file and therefore all of the rules and customizations that are part of XML standards are also applicable to a WSDL file. The WSDL file has the same structure as a XML file. The main purpose of a WSDL file is to define and describe a web service. The WSDL file defines the location of a web service and the operations (methods) that are exposed by that service.

3.2 Software Development Using SOA

Service-oriented software is one of the most evolutionary approaches to software development ever. Service orientation gives enormous flexibilities in developing software: it allows organizations to rapidly and dynamically form new software applications to meet changing business needs, thus alleviating the software evolution problems that occur with traditional applications. There are many problems associated with current methods of building software. The current software is tightly integrated and grows vertically one layer above the other. The ultimate promise of a SOA based software development scenario is the automated construction of service-oriented software, and the chance to discuss software in the context of service orientation and to identify potential problems [7]. Of the many present day possibilities with software development using services is software component reuse; a develop-once and use-forever scenario [9].

CHAPTER 4

SEMANTIC WEB AND ITS TECHNOLOGIES

4.1 The Semantic Web

The semantic web, in its entirety, is an entity that does not operate in isolation from, but in collaboration with, the world-wide web. It is an integrated part of the internet that gives well-defined meaning to data and allows various systems to operate intelligently on the data. Semantic web technologies are presently in a developmental state, and a very small portion of the data has been given meaning using XML-based technologies so that computers can operate on this data without human interference to draw conclusions and perform the necessary actions for the users.

The world-wide web is universal, and the data in it is presently for the consumption of a variety of systems, the purpose being to display it to the end user. The data is variable and is presently specific to the technology that operates on it, and as such it differentiates between the systems that operate on it. One of the aims of the semantic web is to make data generic, such that various systems can operate on the data. To make the data generic, some widely agreed upon technology such as XML is used; most of the technologies in the semantic web are XML-based. The other essential part of such a web would be a rule-based system to operate on the data so that automated reasoning can be performed. Artificial intelligence (AI) has been in existence for a long time, and the AI researchers have come up with innovative solutions that allow AI modules to be plugged into various systems. The present AI systems allow a wide variety of inference and rule-

based operations. Advanced operating algorithms can also be implemented by using these AI systems. The ultimate conclusion of the expanding semantic web would be to realize the web as a single, inter-linked, global system.

Current systems operate in a centralized way; a central processing mechanism controls most of the data and flows for the entire web around it. This centralized control of the web is somewhat restrictive and results in an unmanageable system when the system grows in size. Scalability is a desired property for most of the systems. These systems, as they grow larger in size and complexity, somehow get limited in reliability and provide answers reliably to fewer scenarios. To deal with such a situation, the present systems provide a specialized set of rules so that such scenarios can be addressed reliably. The semantic web accepts that paradoxes and unanswerable questions are unavoidable in versatile systems and focuses on improving versatility. The aim is to make the language of the web as expressive as possible so that a variety of engines with a fixed set of rules can operate on this set of information. The paradigm is shifted from processing logic to processed information (i.e., the data). A mechanism to increase the expressiveness of the data in such a way that it is locatable is one of the major aims of semantic web researchers. Such a mechanism would provide a reliable way for the systems operating on the data to locate the data. The expressive power of the system has made vast amounts of information available, and search engines (which would have seemed quite impractical a decade ago) now produce remarkably complete indices of a lot of the material out there. The challenge of the semantic web, therefore, is to provide a language that expresses both data and rules for reasoning about the data, and that allows rules from any existing knowledge representation system to be exported onto the web.

Adding intelligence to the internet by using systems that make inferences based on existing rules is the task before the semantic web community. The problem before the community involves a mixture of mathematical and engineering decisions.

The logic must be powerful enough to describe complex properties but must also be understood by the rule-based software system. Such existing languages as Resource Definition Framework (RDF) provide suitable solutions to this problem. Minor modifications to existing languages can help make a new language that operates on the data of the web.

The real power of the semantic web will be realized when programs are created that are able to collect web content from diverse sources, process the information, and exchange the results with other programs. The effectiveness of such software agents will increase exponentially as more machine-readable web content and automated services (including other agents) become available. The semantic web promotes this synergy; even agents that were not expressly designed to work together can transfer data among themselves when the data come with semantics.

An important facet of agents' functioning will be the exchange of evidence written in some ontology describing language of the semantic web. With such a language in existence, one can locate a service to satisfy the functionality and then verify the results given out by the first service by making use of other agents, which would use the ontology language used to describe the result. The layers of the semantic web are described in Figure 3.

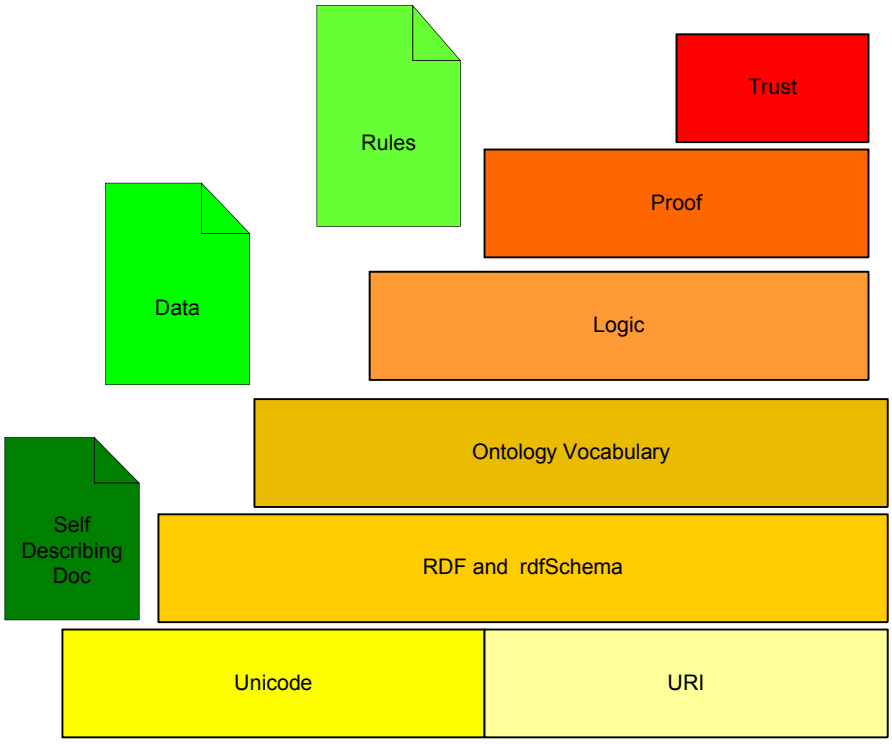

Figure 3. The semantic web layers as described by Tim Berners-Lee

The major application of the semantic web has been in life sciences, where it is used to integrated ontologies from various medical disciplines. Many other disciplines are adopting what began in the life sciences. Environmental scientists are looking forward to integrating data from hydrology, climatology, ecology, and oceanography [10].

4.2 Progresses in Semantic Web

The semantic web shifts the emphasis from documents to data. Much of the motivation for the semantic web comes from the value locked in relational databases. To release this value, database objects must be exported to the web as first-class objects and therefore must be mapped into a system of Uniform Resource Identifiers (URI). Many languages have evolved for the purpose of providing meaning to the data; RDF and Web

Ontology Language (OWL) are the most popular among these languages. RDF has so far had two specifications, the later of which came out in February 2004. This took the basic RDF specification and extended it to support the expression of structured vocabularies. It has provided a minimal ontology-representation language that the research community has widely adopted. RDF has gained popularity, and the need for repositories that can store RDF content has grown. Some focus on providing a rich means to reason over the RDF (Jena is one such mechanism, which is discussed in the next section); while others focus on storing large quantities of data (Oracle's RDF database is one such example). As the stores themselves have evolved, the need has arisen for reliable, standardized data access into the RDF they hold. The SPARQL language, now in its final review stages for the World Wide Web Consortium (W3C) recommendation status, is designed to fulfill this requirement.

4.3 Semantic Web Technologies

4.3.1 Resource Definition Framework

RDF is a language for representing information about resources in the internet and is an extension of XML. Its purpose is mostly to present metadata about resources on the web, such as the availability schedule for some shared resource on a network. RDF is intended for situations in which this information needs to be processed by applications, rather than being only displayed to people. RDF provides a common framework for expressing this information so it can be exchanged between applications without loss of meaning. Since RDF is a common framework, application designers can leverage the availability of common RDF parsers and processing tools. The ability to exchange

information between different applications means that the information may be made available to applications other than those for which it was originally created [11]. RDF is based on the idea of identifying things and describing resources in terms of simple properties and property values. This enables RDF to represent simple statements about resources as a graph of nodes and arcs representing the resources and their properties and values.

RDF is indeed quite simple at its core; though it can become difficult in short order. It is a model of statements made about resources. A resource is anything with an associated URI. This simplicity and uniformity make RDF statements generic. They can be used to encode the above natural-language statement, as well as, say, an object-oriented model.

RDF allows for the expression of such statements in a formal way that software agents can read and act on. It lets us express a collection of statements as a graph, as a series of (subject, predicate, object) triples, or even in XML form. The first form is the most convenient for communication between people, the second for efficient processing, and the third for flexible communication with agent software.

An example RDF document is shown in Figure 4. The first document element, *rdf:RDF*, tells an RDF parser that the child elements can be interpreted as RDF constructs. The first one defines the core namespace for RDF constructs. The second is a special namespace that is controlled by the controlling web organization.

The first child is an *rdf:Description* element, which tells the RDF parser that we have a resource to describe. The *rdf:about* attribute notes the URI of the resource being

described. In this case, it's a local resource to the community site that refers to the overall topic of snowboarding (according to some agreement the community will have made).

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns="http://www. xyz.org">
<rdf:Description rdf:about="http://www.w3.org/TR/rdf-syntax-grammar">
  <ex:editor>
    <rdf:Description>
      <ex:homePage>
        <rdf:Description rdf:about="URL">
        </rdf:Description>
      </ex:homePage>
    </rdf:Description>
  </ex:editor>
</rdf:Description>
</rdf:RDF>
```

Figure 4. An example RDF document

The ideal scenario in a web community would be that each web page maintaining its own metadata. The RDF specification provides a convention for people to place RDF within HTML pages. The empty *rdf:about=""* attribute is a special URI convention that refers to the current document. Other than that, the code is similar to that in the RDF directory. Note that this data can be maintained in tandem with regular HTML <*meta*> tags to support existing search engines and RDF agents. One hopes that vendors of popular web authoring tools will soon produce products that automatically represent metadata in both RDF and <*meta*> formats.

4.3.2 Web Ontology Language

OWL is a language for defining and instantiating ontologies in the web. Ontology is a term that is used for describing the kind of entities in the world and their interrelations. In computer science, an ontology is a data model that represents a domain and is used to reason about the objects in that domain and the relations between them. OWL is more of a markup language for publishing and sharing data using ontologies on the internet. OWL is an extension to RDF and a derivative of DARPA Agent Markup Language (DAML), which started in August 2000 and as is similar to OWL in its motive. OWL describes various objects and their relationship with other related objects in such a vocabulary that it is understandable for a software system. The OWL specification is maintained by the W3C. OWL is seen as a major technology for the future implementation of a semantic web. OWL was designed specifically to provide a common way to process the content of web information. OWL is written in XML and hence can be exchanged among systems implemented in different software platforms. OWL's main purpose will be to provide standards that provide a framework for asset management, enterprise integration, and a language for the semantic web. OWL was developed mainly because it has more facilities for expressing meaning and semantics than XML, RDF, and RDF Schema (RDFS), and thus OWL goes beyond these languages in its ability to represent machine-interpretable content on the web. OWL currently has three flavors, OWL Lite, OWL DL, and OWL Full. These flavors incorporate different features, and in general it is easier to reason about OWL Lite than OWL DL and OWL DL than OWL Full.

One advantage of OWL ontologies is that a wide variety of tools already exist that can understand OWL. The tools available will provide support to OWL ontologies in all domains. Many groups would be involved collaboratively in the construction of ontologies for the internet. The tools available for processing OWL will help an organization that has developed ontologies for its resources.

4.3.3 Jena Inference Engine

Jena is an API in the Java programming language for the creation and manipulation of RDF graphs. It implements the interpretation of RDF specifications. Jena has two primary purposes; to provide an API that is easier for the programmer to use than alternative implementations, and to conform to RDF specifications.

Jena was first released in 2000 and was followed by Jena2 in August 2003. The main contribution of the first version of Jena is the rich Model2 API for manipulating RDF graphs. The Jena API for RDF had many capabilities, such as parsing the RDF, querying RDF documents, and generating input or output for RDF-based systems. Using the API the user can choose to store RDF graphs in memory or in persistent stores. Jena had also the capability of manipulating DAML+OIL. Jena's second version provides additional functionality for general developers to support RDFS and OWL. There are new APIs for accessing ontologies and processing vocabularies. Jena2 also offers two new extension points to system programmers: the first allows the flexibility of developing new APIs to developers, and the second allows the development of new triple sources, particularly of virtual triples that are generated dynamically as a result of some processing, such as inference or access to legacy data sources.

The heart of the Jena architecture is the RDF graph. This layer, following the RDF abstract syntax, is minimal by design; wherever possible, functionality is done in other layers. This permits a range of implementations of this layer. The Jena2 (second version) architecture supports a fast-path query that goes all the way through the layers from RDQL (RDF Query Language) at the top right through to an SQL database at the bottom, allowing user queries to be optimized by the SQL query optimizer. Some inbuilt implementations of the graph layer are provided with Jena2 give a variety of concrete triple stores and some built-in inference, specifically for RDFS and for a subset of OWL. Jena2 maintains the Model API from Jena1 as the primary abstraction of the RDF graph used by the application programmer. This gives a much richer set of methods for operating on both the graph itself (the Model interface) and the nodes within the graph (the Resource interface and its subclasses). Java's single inheritance model is sidestepped to provide polymorphic objects within the layer. This allows the multiple inheritances and typing of RDFS to be reflected in Java [12], [13].

4.3.4 Jess Rule Engine

Jess is a rule engine and scripting environment written entirely in Java programming language at Sandia National Laboratories in Livermore, CA. Using Jess, one can build Java applications that have the capacity to reason by using a declared set of rules. Jess is small, light, and one of the fastest rule engines available. It is suitable for cases where the scope of rule-based reasoning is smaller and where performance constraints exist. Jess uses an enhanced version of the Rete algorithm to process rules. Rete is a very efficient mechanism for solving the difficult many-to-many matching

problem. Jess has many unique features, including backwards chaining and working memory queries, and of course Jess can directly manipulate and reason about Java objects. Jess is also a powerful Java scripting environment, from which one can create Java objects, call Java methods, and implement Java interfaces without compiling any Java code [14].

CHAPTER 5

CASE STUDY: A PROPOSED FRAMEWORK - SOA, SEMANTIC WEB, AND RULE ENGINES

5.1 Implementing Software Components Using Web Services

As discussed earlier in section 3.2, SOA framework is seen as a new methodology for software development. An application can be designed that would implement all of the functionalities using services. An enterprise application is among the most popular areas for which software is currently being developed. In fact, web applications and e-business-related software investments for enterprise accounts for nearly 30 to 50% of all information systems spending [15].

All enterprise applications would have common functionalities that would get built over and over for all of the applications. The modularity and reusable concepts of Sun Microsytems' Java 2 Enterprise Edition (J2EE) framework and Microsoft's .Net framework have significantly reduced the effort expended in developing the same set of functionalities. Component software technology, as exemplified first by Common Request Broker Architecture (CORBA), and then by Enterprise Java Beans (EJB), was a step ahead in the reusability of already developed applications. Various middleware technology products such as BEA Tuxedo allowed reuse of existing software infrastructure developed in legacy languages such as C and assembly language. SOA is another step towards the same end and is much more popular and standardized. The typical scenario would involve a company developing web services and using them

within their intranet so that many applications would share common components. When there is a robust security infrastructure in place and a mechanism for establishing trust exists among organizations, the same set of services will be shared among partner organizations. The application that is described in the following sections assumes that such mechanisms and collaborations among the organization's groups and partners are already in place. The scale of effectiveness of the described framework depends upon the number of recurring functionalities that are implemented as services.

Recently in the enterprise environment, concepts such as single sign-on (SSO), which address the cross-platform security concerns of an organization, have made it possible for diverse applications to make use of existing infrastructures. In a typical enterprise applications environment, the most commonly used functionalities that can be implemented as reusable components are the utility modules, including authentication systems, messaging systems, data validators, and error logging systems. The other reusable components can be the database exposing components, template engines for look and feel, and search modules.

Since there is repeated use of these functions in almost all enterprise applications and, moreover, since the business functionalities tend to remain identical for most of the applications within the same organization or the organizations operating in the same domain, it would be more profitable to adopt a component-based approach. A component-based approach would involve building possible functionalities as reusable components, which would follow the basic rules of object-oriented programming, such as encapsulation and polymorphism.

As an example of such a framework, let us briefly analyze what goes into implementing such components by describing how to build a few essential aforementioned functionalities as services.

5.1.1 Authenticator Component

Authentication systems can be loosely described by the following scenarios.

- An anonymous user can access the unsecured parts of an application without authenticating himself.
- When a user tries to access a secured part of an application, his identity, if existing in the session or application memory, is propagated to the authenticator, which makes a decision depending upon his permissions.

If the application is a web application, the session is maintained by an exchange of cookies, which are unique identification strings in the user's machine. If a particular user is accessing the application from his browser, the application would identify him using a cookie. A web application can exist in one of the following states with respect to security:

- There is no session between the user and the application, and the user has not yet supplied his credentials.
- There is a user session with the application but the user still has not supplied his credentials; this is similar to accessing an unsecured part of the application.
- There is no user session with the application, but the user has supplied his credential. This is the state in which the user has just authenticated himself to the application: he is no longer in the anonymous state, but he yet to start any interaction with the application.

- The user has authenticated himself and is in a session with the application.

The authentication system of the framework, shown in Figure 5, satisfies the aforementioned requirements.

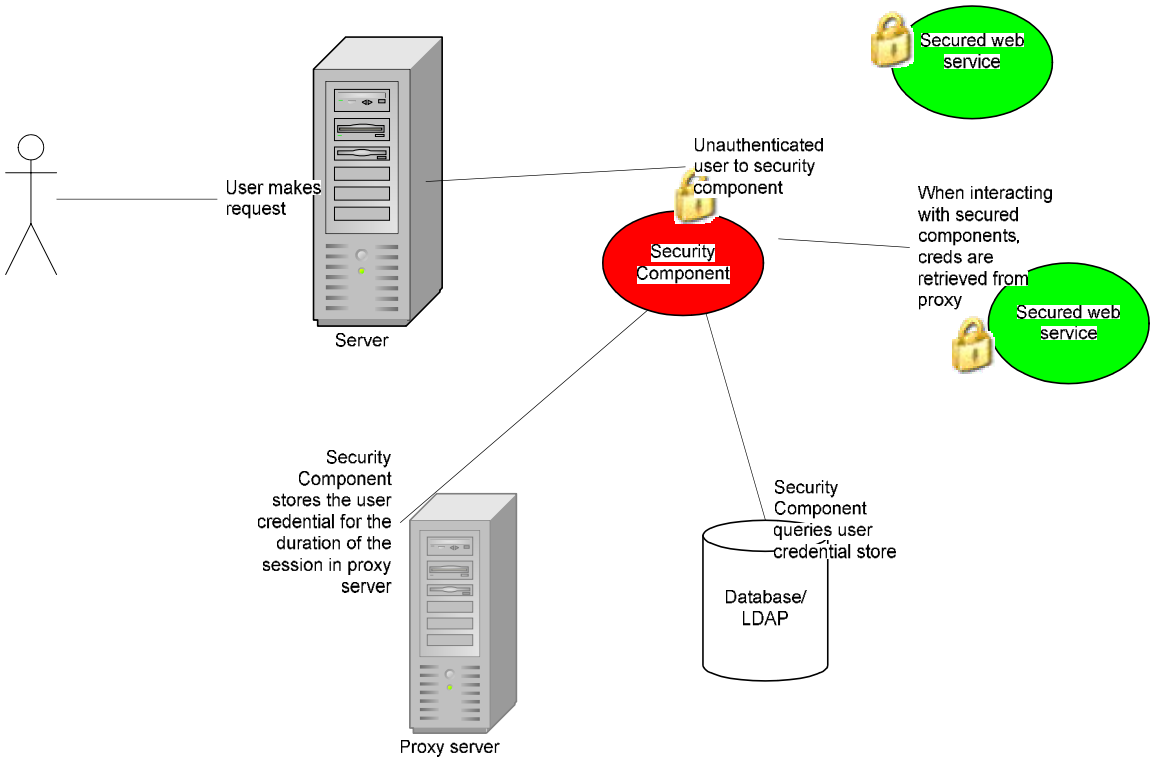

Figure 5. The authentication system of the framework

A web-services-based system would have security requirements that are a superset of a web application. For instance, in most of the web applications, user identity sessions can be maintained using HTTP cookies. Web-services-based systems are expected to have components that are built on different platforms, and maintaining a session using cookies might not be possible for multiple platforms. The platform-independent way to establish a session would be to use XML signatures. Since there might be different physical domains in which the components would be located, the user

should be authenticated to various other applications within the enterprise using a single set of credentials. The security component should allow various modes of signing-in the user and mapping his credentials across the domains where the components are based. The variety of authentication supported would include such means as digital certificates, userId – password, and tokens. The security component would also be able to query a variety of user databases, including Lightweight Directory Access Protocol (LDAP) based and Relational Database Management Service (RDBMS) based databases, to verify user identity. For all the deployed web services that are registered, the system would provide centralized authorization and access control services. The assertion message exchange would be added to the underlying SOAP-based messages by which web services normally communicate. The architecture would be similar to a proxy-based system in grid computing [16], where after verification of the user's existence his proxy credentials would be stored in the local system, so that the user can be identified locally for the lifetime of his interaction with the system. This would allow for the effective implementation of SSO in the system [17].

5.1.2 Search Component

The search engine is one of the most commonly used functionalities and is normally a part of every web application. Building a versatile search component, which takes in parameters as inputs and returns the results, would help in greatly implementing code reuse. The search component would be optimized to perform faster searches. As a result the performance of all of the applications that use the search component would be improved by performing optimizations only for the search component.

The search component would allow the application to specify a variety of search targets. The targets could be databases, flat files, or a machine's file structure in the corporate intranet. Since the user would authenticate himself using the authentication component described in the preceding section, which would use the SOAP message mechanism internally to send out requests to other components, the search component would expect the user's identity to be encrypted with the user's request message (in most cases the user's request might also be encrypted by using his private key). The search component would decrypt the request using the user's public key. This mechanism of authentication holds good for the rest of the component. For a discussion on authentication using certificates, please see Appendix A.

The search component would expose itself using a WSDL as per the web services standard. The WSDL would contain information about the search component, such as the unique name of the service, the URL, the types of search input parameters, the type of output to be expected, the search targets, the binding name, and the endpoints.

The search input parameters would be normal words or characters; numbers would also be allowed in the search input text. The input can be a combination of words, single words, or strings. The special characters would be ignored, as they cause errors in processing the search.

The results of the search can be returned in a variety of forms. The search component would either generate files in the specified format and save them in a common location or send the results back to the calling component in XML format [18].

The main structure of a WSDL document looks like the listing in Figure 6.

```
<definitions name="IntraSearch" targetNamespace="urn: Search"          xmlns:typens="urn:
Search" xmlns:xsd="http://www.w3.org/ "
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
<types>
<xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="urn:Search">
    <xsd:complexType name="SearchResult">
     <xsd:element name="documentFiltering"        type="xsd:boolean"/>
      <xsd:element name="searchComments"           type="xsd:string"/>
      <xsd:element name="estimatedTotalResultsCount"  type="xsd:int"/>
      <xsd:element name="estimateIsExact"          type="xsd:boolean"/>
      <xsd:element name="resultElements"           type="typens:ResultElementArray"/>
      <xsd:element name="searchQuery"              type="xsd:string"/>
      <xsd:element name="startIndex"               type="xsd:int"/>
      <xsd:element name="endIndex"                 type="xsd:int"/>
      <xsd:element name="searchTips"               type="xsd:string"/>
      <xsd:element name="directoryCategories"  type="typens:DirectoryCategoryArray"/>
      <xsd:element name="searchTime"               type="xsd:double"/>
     </xsd:complexType>
    </xsd:all>
   </xsd:complexType>
```

Figure 6. The example WSDL for the search component

5.1.3 Database Component

Exposing a database as a web service would form another important component in a web-services-based system environment. The database can then be accessed by other web services in the environment, and the needed information can be queried and retrieved from the database. In the case of updating the database with information, the web service would take in the necessary information and, after proper authorization, insert it into the database tables.

The basic architecture of the database web service component would be a web service layer in the front that exposes itself to the network, a Java-based layer, and a database in the backend that communicates with the Java-based layer using present database connectivity techniques, such as Java database connectivity (JDBC). It is possible that an environment could have numerous databases, which would be from different vendors and hence be different in their properties (e.g., Oracle 9i, Sybase, and IBM DB2), each of them could expose itself using a web service so that an application or another web service in the network would implicitly access it depending on the data it needs. The database components would then enlist themselves using web services registries and other such optimizations as discussed in later chapters. In our case, to prove that such a collaborative environment could exist, we take a simpler case of just one database component.

The exposed web service module can be implemented using J2EE or .NET framework, which are the currently, most common ways of implementing a web service. More options for implementing web services using any programming language should be available in the near future as web-services-based technologies evolve. Communication

with external services will be achieved using SOAP, as described previously. Since the amount of data that needs to be transferred would be sizeable in most cases, SOAP with attachment is the best option available. SOAP with attachment uses SOAP envelope and an encapsulation technique, such as Direct Internet Message Encapsulation (DIME) or Multipurpose Internet Mail Extensions (MIME) data. The different parts of the message can be unassembled by the receiving application. As per the W3C specifications of SOAP with attachments in RFC 2387, the following rules would apply for SOAP with attachments depicted in Figure 7.

- The primary SOAP 1.1 message must be carried in the root body part of the multipart/related structure. The type parameter of the MIME will be same as the body type of the rest of the SOAP document, i.e., XML or text.
- Referenced MIME parts must contain either a Content-Id MIME header structured in accordance with RFC 2045, or a Content-Location MIME header structured in accordance with RFC 2557.

The security can be implemented in a manner similar to the normal application security implemented in J2EE components, such as encrypted keys in a web service environment and userId-password based authentication in normal applications. The web services layer would decide on the authorizations of the user and would attach a role to it so that authorization decisions would be possible in the other layers.

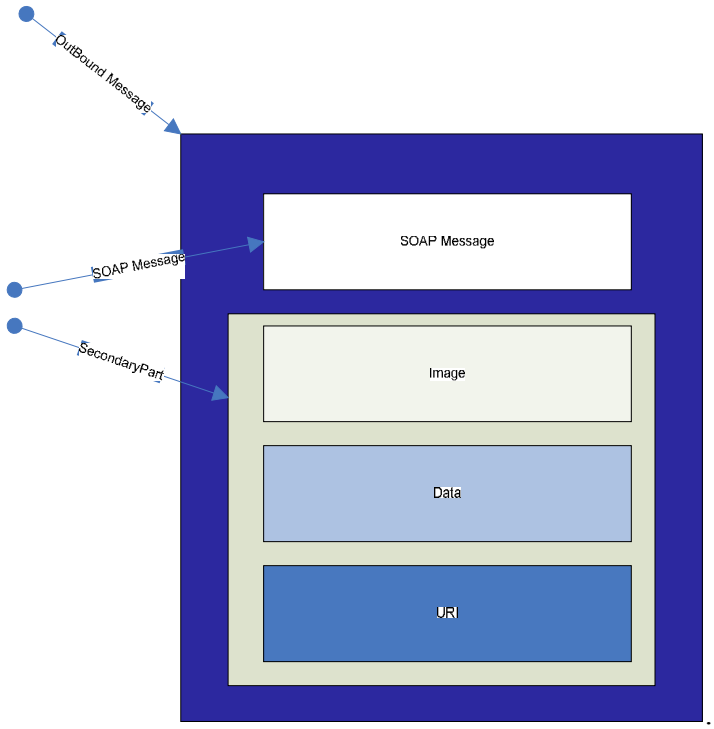

Figure 7. SOAP with attachments

In the search component, the web services layer would also process the information in the SOAP messages and would pass it in a processable format to the Java layer. Processing the information would involve deciding on the action the interacting component or application wants to perform on the database. This might be the normal *create-read-update-delete* operations of the database or a request for the metadata of the schema structure.

The Java layer would take in the user inputs from the web services layer, which would then be passed as processable string parameters in relevant method calls to the Java layer. The Java methods would use these parameters and make a call to the database using either JDBC or messaging services. The messaging services implemented in Java that form a part of J2EE are called Java message services (JMS). The JDBC calls would

use prepared statements or queries as applicable. Wherever the queries are expected to take a greater return time, messaging techniques would be used to send the queries to the database. There could be a scenario in which the same database is being used by system-level legacy applications at the same time; messaging can be an effective way of dealing with scenarios in which multiple interactions would be handled asynchronously. The Java layer would avoid complicated business logic; such complex interaction with the database would be taken care of by stored procedures in the database layer itself. The Java-based layer would only make calls to such preexisting logic when required. Such complex business logic would even be exposed through the web services' descriptors directly so that clients could make calls directly to get desired results. Finally the results returned from the database layer should be converted to XML format and returned to the web service layer, which would then put it in the SOAP response message and send it to back to the calling component or application.

The database layer would be any normal relational database, which includes such popular relational databases such as Oracle and Sybase. The database itself need not have any specialties except for supporting XML and XML queries. XML Database (XDB) is a database feature in Oracle (Oracle 9i Release 2) that supports XML data storage and retrieval. It also offers manipulation of that XML data. Standards such as XQuery and XPath are aimed at supporting XML in the database.

Many schemas in a database can be exposed by a single web service front end. The database information can be configured in an XML file that is readable by the web service client. Web services assembler products such as those from Oracle (Oracle Enterprise Manager) help in automating this process. Once the web services database

components are packaged and deployed in the intranet, all eligible applications are able to see the information published using WSDL.

5.1.4 Logging Component

Logging is the final element in the list of functionalities that would be discussed in implementing web service components. Logging code is present in almost every method (assuming that the programming methodology is Java) in all of the modules of a system. If logging is implemented as a component, the overhead of implementing logs for each method can be delegated to it. The application would simply register itself with the web service component and define its log messages in a commonly readable file; the details of implementing and maintaining logs would be taken care of by the component. There are currently many utilities that allow logging code built into the system, with per package level control of log data generation. Apache's Log4J is one example of a J2EE platform logging utility, and the Apache Avalon project's Logkit is another. The logging component would be built on similar lines and would be configurable; it would be support multiple configuration mechanisms that are determined by the platform it is set up in.

For the web service components in general there would be management interfaces. This would be a Java Management Extension (JMX) for Java-based web services or a Windows Management Instrumentation (WMI) service interface for .NET framework services. Such management mechanisms allow for the control of the web services' behavior. The variety of control that would be allowed by using management interfaces includes the setting of attributes such as control access, measure data about the

access to the web service, monitor the way web service accesses other critical resources, and measure the performance of the web service.

5.2 Implementing Service Registries and Inference Engines for Locating Services

Inference engines were discussed in some detail in the previous chapter. The purpose of including an inference engine in the present framework is to use it in identifying and dynamically federating web services. The concepts of dynamic web services federation will be discussed in detail later in the chapter. In this section we discuss how inference engines can be used to intelligently locate the needed services when we configure a service definition in it.

Before we discuss how the inference engine can be used in this framework, a brief discussion about how UDDI would expose services in the intranet is necessary. UDDI for a particular group of service would expose information about the entity and its APIs. These registries would be run by multiple service groups and would be used by the groups within the organization or partner organization who would desire to share their web service entities, as well as by the inference engine layer of the system that would contain the links to all these registries in order to find the desired service.

In order to prepare an application to take advantage of a remote web service that is registered within the UDDI registry by other businesses or entities, the application needs to use the information found in the registry for the specific service being invoked. This type of inter-business service call has traditionally been a task that is undertaken during the development of the application. By implementing UDDI registries, this will

not necessarily change completely, but one significant problem can be managed if a particular invocation pattern is employed.

The web service groups would use category codes provided by UDDI to publish the service; for example, a specific category code such as *DBEMP* will be used for the employee database component group. These would classify the web services in that group of the registry as the ones that expose the employee database. The web services components layer which would make use of the UDDI API for adding the classification codes for the web services.

The *bindingTemplate* data obtained from the UDDI registry represents the specific details about an instance of a given interface type, including the location at which a program starts interacting with the service. The inference engine, upon finding this service would be able to create a new RDF document for the web service component that would store the name and properties of the service for future use. This information would be stored in a local cache. If a service is moved from the location, the data stored in the cache would be refreshed. In this case the inference engine layer should make a fresh call to the UDDI registry and update the previous RDF with the web service information. Since this would significantly affect performance, it is assumed that, web services would very rarely be moved to new locations (meaning that the URI should change rarely).

By using this pattern with web services and the inference engine, a business using a UDDI operator site can automate the process of web services discovery and choreography without significant overhead in the invoking application. The application side, as mentioned earlier, would be automated using inference engines and the

technologies of the semantic web, such as RDF. Effort would be involved is coding the application once and enhancing or modifying the application to support a new group of requirements. The inference engine layer would then be used to locate services and theoretically be used with all kind of front-end applications that would send as input the functionalities they need for a particular use-case and in return would receive a composition of services to satisfy that need. For this, an intelligent inference engine layer that would require minimum human interference and would work automatically needs to be built [21].

The Jena Inference engine from HP Labs, which was introduced in the previous chapter, is the inference engine that would be used for the framework. For the application being discussed, an RDFS RuleReasoner would be used as the reasoner for the model data which would be represented in RDF. The RDFS RuleReasoner would be extended and customized for the inference engine layer to operate in a web services environment. The set of RDFS RuleReasoner Java classes that would be created would be SearchReasoner, LoginReasoner, DatabaseReasoner, LogReasoner, GenericWebServiceFinder, and GenericWSCommunicator. Each of these classes would be assigned the function of reading the RDF model data, which would be used to model the four core functionalities of the application (i.e., login, search, log, and database access) and making decisions on which web service to call. The GenericWebServiceFinder would be responsible for associating with a particular web service, and the GenericWSCommunicator would perform the communication with the web services on behalf of all of the functionalities in the application.

A reasoner would be attached to the RDF documents that are specific to the web service groups. An RDF would be a model from which the built-in reasoners would read and process the application logic. The reasoners would then derive logic from this data to create an instance RDF; the web services group information would be RDF model data, whereas the information about a specific web service would be instance data.
The RDFS RuleReasoner would be configured to work in the full mode. This would implement all of the RDFS axioms

There are times when the data in a model bound into an InfModel can be changed "behind the scenes" instead of through calls to the InfModel. If this occurs, the result of future queries to the InfModel would be unpredictable. To overcome this and force the InfModel to reconsult the raw data, the *InfModel.rebind()* call would be used. This reasoner would be accessed using *ModelFactory.createRDFSModel* or manually via the method call *ReasonerRegistery.getRDFSReasoner()*.

The RDFS RuleReasoner would be a hybrid implementation. The subproperty and subclass lattices would be eagerly computed and stored in a compact in-memory form using the TransitiveReasoner. The identification of which container membership properties (properties like *rdf:1*) are present would be implemented using a preprocessing hook. The rest of the RDFS operations would be implemented by explicit rule sets executed by the general hybrid rule reasoner. The three different processing levels would correspond to different rule sets. These rule sets would be located by looking for files *"etc/*.rules"* on the classpath and so could, in principle, would be overridden by applications wishing to modify the rules.

RDF would be used to describe web services locally in the inference layer. RDF would exist as XML documents in the inference engine. A web service group or category would be described as a resource in the RDF sense. The UDDI registry path would be the URI of the resource; all web services belonging to the same domain, group, or category would be represented by the same UDDI registry. For example, the *WSDL* property has value that would point to the URI of the description, and the name property of the web service has value equal to the web service published name.

Once a web service is discovered, a RDF model would be prepared by using the Jena RDF writer classes. This RDF document would encapsulate the details of a web service and would be the point of reference when similar calls are made to the web service in the future. A sample of how to create a model RDF is shown in the code block in Figure 8.

```
Public static void rdfDBEmpWriter(String args[]) {
       // some definitions
        String webServiceEP    = "http://somewhere/JohnSmith";
        String wsName    =  "DBWebService";
       // create an empty model
        Model model = ModelFactory.createDefaultModel();
       // create the resource
       //   and add the properties cascading style
        Resource dbEmpWebService
          = model.createResource(webServiceEP)
                 .addProperty(WS.NAME, name)
                 .addProperty(WS.DESCR,
                              model.createResource()
                                   .addProperty(VCARD.Given,
givenName)
                                   .addProperty(VCARD.Family,
familyName));
       // now write the model in XML form to a file
        model.write(System.out);
    }
```

Figure 8. Sample Java code to create RDF model

The resulting RDF model would be similar to the one shown in Figure 9. The Jena RDF Reasoner for the database group would process the semantic data about the related web service group and would decide which web service to call, depending on the scenario. Each of the RDF models of a web service would adequately represent a particular web service so that the inference engine calls the correct web service when the criteria described in the RDF model is met. The suitability of the web services would be based on the decision made by the RDFS Reasoner and would be as per the functionality that is requested by the interface layer.

```
<rdf:RDF
  xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
  xmlns:vcard='http://www.w3.org/2001/vcard-rdf/3.0#'
 >
  <rdf:Description rdf:about='http://context/WS'>
    <ws:Name> </vcard:FN>
    <vcard:N rdf:nodeID="A0"/>
  </rdf:Description>
  <rdf:Description rdf:nodeID="A0">
    <vcard:Given>John</vcard:Given>
    <vcard:Family>Smith</vcard:Family>
  </rdf:Description>
</rdf:RDF>
```

Figure 9. The RDF model of a web service

5.3. Integration of the Layers: Service Integration Using the Intelligent Framework

In the previous sections of this chapter, the individual components of the framework were discussed. In this section, the application is presented as an integrated entity with a description of how each layer fits into the overall picture of a tractable software framework where it would be possible to forward and backward map the requirements, architecture, and implementation. Such a system concept would try to

answer some of the problems that are faced in current software engineering practices, specifically the ones described in the first chapter. The example system design shows how a reliable framework with limited complexity can be built (the scenario shown in this case would be a system which has straightforward functionalities); more complex systems would be difficult to implement and would require advanced versions of the inference engines. The overall picture would present the architecture and then describe the interaction of each layer with the other. The system architecture can be described in layers, as shown in Figure 10. The topmost layer, which would interface with the user, will be the user interface (UI) layer. As is the case in most web based or systems applications, the UI layer would allow users to request information and show the results. In the present system it will be realized using standard technologies such as Java Server Pages (JSP), XML, and HTML. There would be different JSPs for different views to be shown to the user for different information requested. Each page would have specific information and would be processed by a related Java Servlet class in the controller layer.

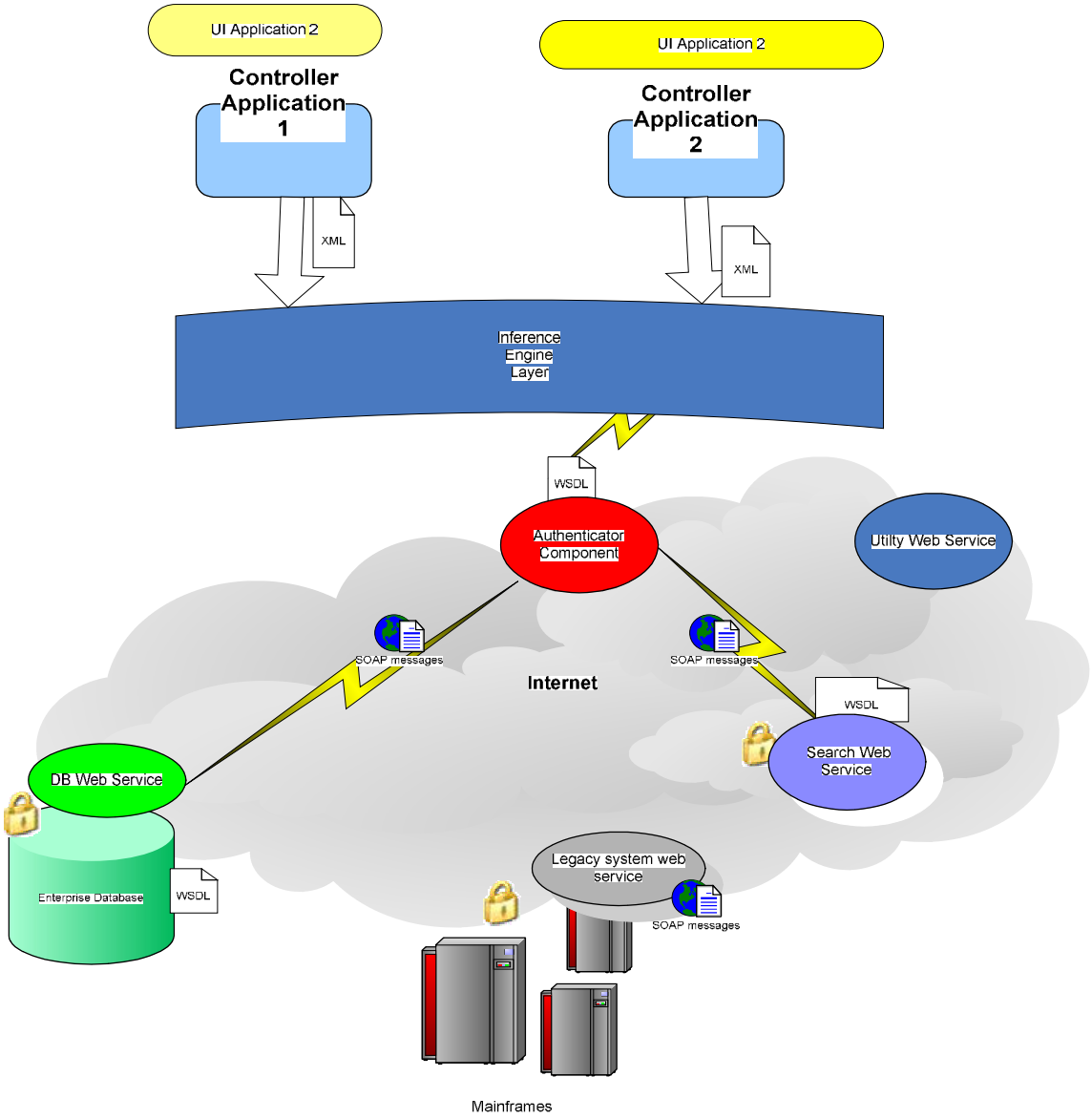

Figure 10. The high level block architecture of the framework

The controller layer would be a group of Java servlets that would each process a specific user request type and would be related to particular JSPs in the UI layer. This group of Java classes in the controller layer would communicate with a corresponding group of classes in the inference engine layer. The servlets would be responsible for taking in the request from the JSPs in the UI later, processing it, and making a decision as

to which inference engine group to call based on mapping rules. It would make a call to the inference engine layer with the user request passed as parameters. This sequence is shown in Figure11. The protocol for communication between the controller and the inference engine would be XML; between the UI and the controller it would be HTTP. Because of this the user would be able to parse XML data using standard XML parsers, such as Java API for XML Parsing (JAXP). The parsers in the controller would be mostly Document Object Module (DOM) parsers and would, as a result, parse whole XML documents. The decision for the parser type is made under the assumption that the data passed between the layers is not very large in size. The results passed onto the controller from the inference engine as a result of user queries would not hinder the performance of the controller under such situations. The controller would make use of special APIs to put in the user's requests parameters to the XML defined using a standardized Document Type Definition (DTD) for the system. The DTDs for the system would define XMLs for passing in requests and for getting results. The XML format for errors and other scenarios would also be defined by the DTD. Similarly when the controller receives result data from the inference engine, it would receive a defined type of XML that would carry the result to be shown to the user and other meta-information.

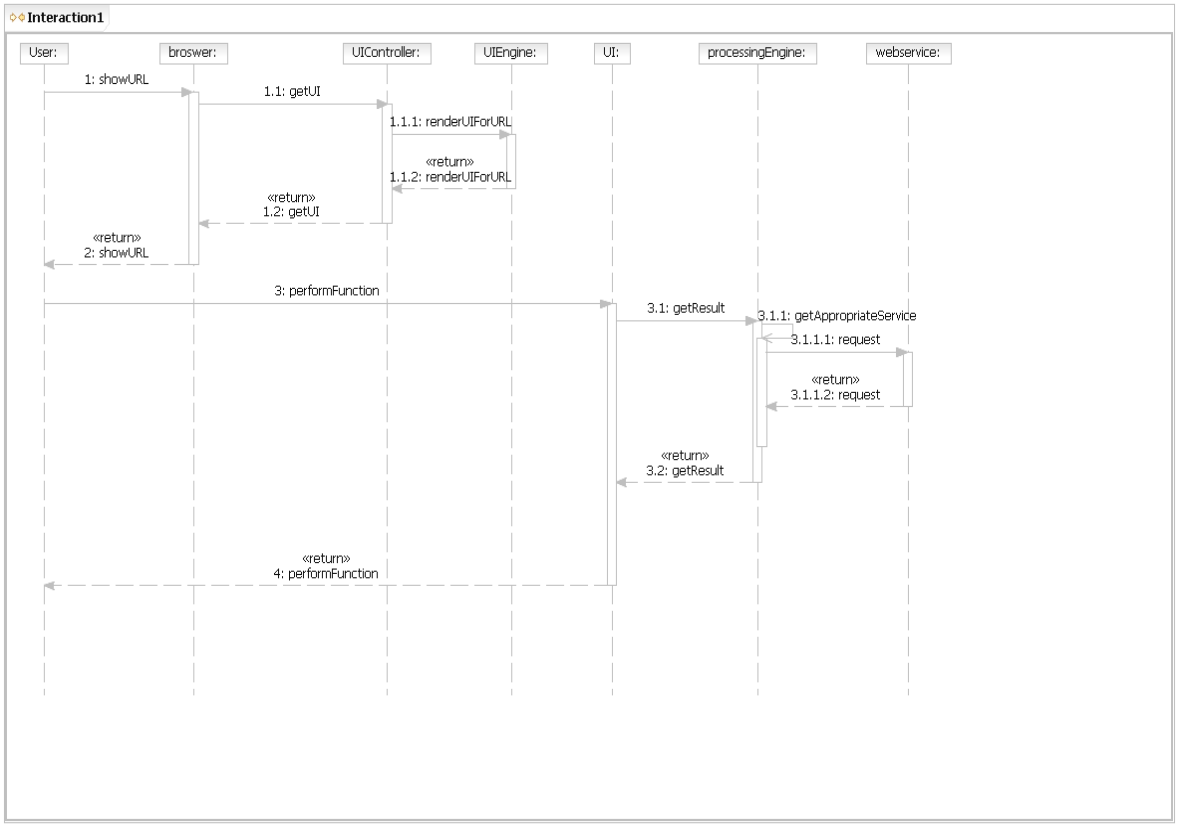

Figure 11. The sequence diagram for the framework

The inference layer would be very much similar to any other middleware layer in an enterprise application except that most of the logic would be implemented using the Jena and Jena RDF API. In doing so, the system would be able to operate without implementing the business logic; the business logic would be implemented at runtime by the artificial intelligence of the inference engines. The XML data received as input from the controller layer would, as discussed earlier, contain the requests from the user. The user request would be satisfied by some existing component. The controller would be responsible for directing to the module in the inference engine which would be responsible for providing the related group of functionality; the service that finally processes the user's requirement is satisfied by the web service that is located by the

inference engine layer. For example, in the scenario of authentication, the user would input his credentials, the controller would redirect it to the authentication module in the inference engine, and the inference engine would parse the input data to make sure exactly which authenticating web service to call, is based on the user's credentials and domain. In another scenario, the search parameters entered by the user to search information in the network would be passed on to the search module of the inference engine layer, which would then locate the services that would be able to perform the search and also the database web service that would provide the data that is to be searched, hence providing a federation of web services to satisfy the user's request. The second scenario describes a situation where two or more web services collaborate to provide service to the user. Such a federated situation would be supported by the inference engine layer; the inference engine would dynamically compose a workflow for the composition of web services.

The final layer is the group of web services exposed by the group's directory services. This group of web services would include the databases exposing their data, the important difference from other enterprise applications being the fact that there would be no database in the backend, but a layer composed of a group of components (this would be effectively the backend) that by themselves provide a complete functionality.

The system would consist of various services that are already developed. The services would be generic enough to satisfy the requirements of the systems. The databases would be also exposed by services.

The next section describes the various advantages that would be offered by such a

system in the area of software development and in dealing with various problems associated with software development.

5.4. Software Development Problems Solved by the Proposed Framework

The various problems associated with the current software development are discussed in Chapter 1. The case study described a system that integrated various technologies and thereby developed a framework that can be used in software development. Such a framework assumes that one could develop web services that are generic so that could be used easily with other systems. The role of such a system and how it would evolve during the software development lifecycle is described in the next paragraph. The solution it provides to development problems is mentioned along with a description of the system.

In the requirements phase, a set of requirements that initiate the software development lifecycle would be elicited from the client by an architect and an domain expert. At the beginning of this phase, the client requirement would be further decomposed into individual functionality requirements and mapped with existing services. In cases where a composition of services might help meet the requirement, the architect would map the requisite services with the requirement. At the end of the requirements phase, a requirements-implementations graph would be created which that map each requirement node with specific components. In a case where the specific node of requirement is not currently implemented by any service, a new service would be written to satisfy that requirement. Since the requirement and components would be explicitly mapped and requirements would be initially decomposed into sub-requirements

nodes, the problem of imprecise communication would be solved. The concept would be similar to existing pair programming techniques, but in this case an architect would be the pair up with the customer. Suitable tools would be used to record these activities and generate the necessary documentation and diagrammatic representations at the later stages. The mapping would also allow the architecture to be traced back to the requirements at later stages, thereby doing away with the problem of intractable requirements and design. The detail to which the requirements are decomposed would ultimately depend on the size of the requirement itself and the components that would be required to satisfy it. Typically the requirements decomposition would be detailed enough to allow the architect to map it to components. The mapping additionally creates a backward mapping, such that in the case of the decrease of scope of the application, the individual components can be ruled out.

In the architecture phase, the architect would consult the graph created in the requirements phase to list the services that would realize the system. The architect would then design the additional services that need to be created for the nodes of requirements that were not mapped to any services. The general approach when building new services would be to design generic services that have a wider scope and would possibly be of use to other systems. The list of services would then be prepared, and the mock-up for the various use cases of the system would be done with the services that were identified to satisfy those requirements. Each of these use-cases would typically need many services, and the federation of various services at various stages of the use case scenario would be worked out. Web services, when operating in composition, should have a workflow; this workflow needs to be implemented in the inference engine layer. The architecture would

also ensure that all of the utility services such as loggers, comparators, and error handlers are common to all systems in the environment and would be implicitly called from within a service whenever needed; these services would not form a part of the workflow. The architect would add the inference rules for the various phases of the use cases if they are not in existence already, so that the system behavior is as expected for a particular input from the user. Because services would be involved in executing all of the scenarios, and because all of the systems in the environment will have a common inference layer as an interface to service layers, the architect would end up adding new rules to the inference layer and new web services to the service layer for those not in existence. This would make the architecture integrated, and the system would scale very well. The problem of unforeseen factors is almost non-existent in this case because of the component approach adopted and also because of the requirements to architecture mapping.

The detailed design phase would involve designing the needed services as per existent standards and also designing the additional rules needed in the inference layer. The services would be designed such that they are generic enough to satisfy future requirements also. For example, the search service component described in the earlier section would accept any type of search parameter as input, search a variety of targets, and output the desired number of results in the desired format; the only condition is that these parameters would be described in XML syntax in the SOAP message. Similarly, all other services would also be designed such that it would be a superset of the present requirements: any future demand of a similar functionality would also be met by the service. Because of the web services approach, the platform dependency and the programming language in which the functionality is implemented would not be a

constraint. The tangling of code blocks in various modules would also be absent because the utility and core functionalities would be built as separate entities, so the clean separation of concerns similar to that promised by aspect-oriented programming can be achieved. The modules in the service would be conveniently mapped back to the architecture components, which would also be very strongly mapped with the individual nodes of requirement. The result is a very tightly mapped system which retains mapping between phases and avoids the problem of traceability in each phase, requirements, architecture, and detailed design.

Finally the effort expended in system testing phase would be greatly reduced because the requirements verification which forms a considerable portion of the system testing phase would not be necessary, as that would already be taken care of in the initial phases.

CHAPTER 6

CONCEPTS IN IMPROVISATION OF A SERVICE BASED ENVIRONMENT

An environment consisting of numerous web services that form a system that operates independently without much human intervention is a focus for many researchers. Among such groups the most noted ones are engaged in intelligent software agent research. An agent is a software component that is able to perform operations in an automated way without any human user interference. A system of agents system is an entity or a piece of software which sends and receives messages and acts on behalf of the web service. An agent is an implementation of web services. The major groups engaged in the research of intelligent agents are the Foundation for Intelligent Physical Agents (FIPA) and the IBM Watson Lab. The Intelligent Software Agents Lab at Carnegie Mellon University is another such body.

The framework described earlier as a case study has the essential ingredients of an agent-based framework, except for the fact that the intelligence is not embedded in the web services; it is a separate layer and forms a different module of the application. Nonetheless there are many concepts that can optimize either kind of environment (i.e., an agent based or a web service based environment) that are discussed in the following sections. Some of the concepts described are at an incipient stage, and there is considerable research directed towards them.

6.1 Dynamically Defined Systems

Let us look more closely at the service framework described earlier. Such a framework negates the concept of a defined system. There is no particular module which by definition belongs to one of the systems in the environment. Moreover, in this environment, there is nothing that separates and completely defines a system with its boundary. The user requirements are satisfied by a federation of web services and some module of the inference engine and user interface layer. In such an environment a system can be only loosely defined. A system is actually constructed on the fly whenever a user makes a specific request. A system that satisfies some specific request would come to being for that request and would not exist in the very next request. We could call such a system a ***system on the fly***, an ephemeral system that exists for the time of the user's request and does not exist anymore upon the completion of that request. The only thing that would be truly defined for the system would be the system's UI layer, built using JSP. It is possible that the user interface layer can also be built at run time using template engines and content management tools. In the near future the concept of a well-defined system might be redundant, and the environment might consist of many short-lived systems. A corporation's IT investments would be directed towards a reusable components environment rather than towards particular applications, as is the case currently.

6.2 Services Clusters

In a web services environment, it might be difficult for a service requestor to identify the particular service that would completely satisfy his or her requirements. In

such scenarios, web services can be grouped together logically and exposed using a common UDDI. A group of related services can be clustered such that, for a related functionality, the application would access the cluster for finding the particular web service. The specific web service that would process the caller's request would be decided by the kind of request the calling application sends across to the web service groups. A cluster's description would be same in concept as that of an interface; it would provide an overall description of the services, and the constituent web services would be specialized implementations. The web services belonging to a cluster might not be physically in the same network, they would probably be deployed in different physical machines. The foremost advantage of a cluster would be that it would provide an aggregation of related functionalities so that web service clients would find the related services without the overhead of searching through all of the published web services. The UDDI for the cluster would expose the entire group of services in it, and hence the clients would be accessing the published information for the group rather than searching individual published web services. The performance would improve more that ever as the overhead associated with accessing the published information of individual web services is eliminated. This kind of clustering would particularly be helpful in a scenario in which partner organizations having web services which provide related functionality would group together these services under one cluster.

6.3 Service Gradation

Of the many web services that might be available to serve the user's request, the calling application may find a few to be very useful. These web services might become

the ones which the application uses most frequently, and it makes sense for an application to cache the location of these web services for use in the future. This concept can be taken further to include a list of web services, none of which would permanently remain in the web services list. A cache would store the initial handshaking parameters which the application would need to establish a connection with the web service. Whenever one of the web services was used less frequently, it would be downgraded by the application and would ultimately be cleared out of the cache. The concept is similar to the J2EE application servers like BEA Weblogic caching policy for session EJB components in their local cache and more similar to operating systems' caching technique. Since the cache would change in its configuration this kind of gradation is dynamic; no web service would remain in the cache for ever, and the contents of the cache would depend solely upon the kind of services the user needs.

Web services in the network would also be sorted out by an application as per its usability. For a particular category of web services (which would all cater to the same functionality), the application would find the accurate results from a very few, and it would associate those web services for that particular functionality. These optimizations would greatly improve the general web services operating environment.

A local configurable and updatable listing of the source of web services will be maintained. The list will be stored in the local database as a table or may be in some other machine readable format as XML. The list will be initially created by the domain experts and will be updated through an interface when there are updates to the list of services. The listing will be also be updated by agents, i.e., the local service agents. The listing will be divided into categories and sub-categories depending on the classification. A new

service will be added to the corresponding category whenever it is discovered. A service which is provided by a partner organization can be in many of the categories if the ontology fits into multiple categories.

CHAPTER 7

UNIQUENESS OF THE APPROACH: A BRIEF COMPARISON

The technologies described in the last chapter can be included in the framework described. One of the features, i.e., dynamic definition of systems, is already ingrained in the framework. Since the framework supports a purely web services based environment, service clustering and grading can be incorporated into individual segments of the framework. The framework would be more efficient with features that generally improve the performance of a web services environment. Web services is presently an area of many innovations and would mature technologically with a wider user base and better products developed for it in the near future. Such a framework which has web services as its primary constituents would profit from any developments in the area of web services. Hence an inherent advantage of such a system, which uses web services and semantic web technologies, is that the scope of its innovation is not limited because it avoids dependencies on legacy technologies; any such systems present in the environment would be exposed by web services hence obliterating the legacy technologies used in the backend.

Software development methodologies that recognize SOA and make use of web services generally use the previously mentioned development methodologies for developing the interface to the web services layer. Even the web services are developed using the older methodologies of software development. In many cases, web services lose their rich nature because of their development with a particular application in view.

Service functionalities are replicated and services become application specific due to this narrow focus during the requirements phase of the project. The proposed framework's focus is opposite in the sense that it recognizes web services as the focal point and develops the application around existing services. Hence the framework can be thought to be more *services-centric* than *applications-centric* in its approach.

Existing frameworks do not support seamless mapping of all phases as the proposed framework does. As described in the beginning chapters, the existing methodologies except RUP do not have any proper requirement management technique. Additional requirements, which creep into the later phases of software development which ultimately creep in most cases result in implementation confusions even in a well managed software development processes. The proposed framework takes into account any kind of requirements and makes the process of requirements management much simpler, since it relies on mapping requirements to components. The simplification of the requirements management process is one of the unique features of the framework.

The other major unique feature of the framework in terms of software development methodology is the well-focused design process that results due to the fact that the design is nothing but a design of components. Architecture and design phases in existing methodologies generally initiate the application design in most cases; the application is built from scratch in most cases. In contrast, the proposed framework has very well-focused design objectives. The design phase focuses exclusively on building service components whose functionalities are well defined and in providing an inference layer wrapper if it is not in existence already. All the inference layers modules would be very similar in their basic design and the individual implementations would vary. This

similarity in design helps in constant improvement of the inference layer design. Optimizations to the inference layer design would be a direct result of this similarity in the modules and would help in cutting down the design time.

CHAPTER 8

CONCLUSION

In the previous chapters, the framework was discussed in detail and the need for such a framework was justified by listing some problems with the current popular methodologies. There are many certified processes and products that are constantly being developed to help solve these issues. Quality control processes are being adopted universally. Whereas processes like Six Sigma [19] were introduced in the manufacturing industry to help improve the quality of products, they are being used widely in the software industry and affect each phase of software engineering. Frameworks have been designed to implement the Six Sigma methodology. Another similar methodology is the Capability Maturity Model (CMM), which in its concept is related to the entire software development cycle [20]. CMM has five defined levels and companies gradually move up the level beginning at Level 1. There are products that support a company to adopt CMM and there are many consulting companies which help in implementing CMM standards. The popular standards of today are being developed with investments and interest from major companies in the software development arena. Most of these standards and frameworks are a result of the existing collaboration in the industry to deal with common problems, and collaborative effort and continuous research is necessary for the sustenance and use of any new framework.

The proposed framework presents a comparatively simplistic scenario of software development world and tries to solve the major problems by assuming the absence of

many of the real world factors. In reality, many problems ranging from conflicting software environments to human factors would pose some problems in implementing such a framework. For example, each of the groups implementing the web services would try to customize the services to meet its own set of problems, rather than putting in the extra effort in making it generic for use with each of the groups. The inference engine layer might be designed and implemented in a non-standard way by the developers. Problems would arise in the automating web services discovery and workflow composition because of irregularities in the implementation of the inference layer. The biggest threat to such a framework would be complex requirements that would be impossible to map to individual components and which might get spread across many components. In such a scenario, the succeeding phases would suffer, and it would be impossible to implement an inference engine which would call the appropriate service due to the present limitations of the current AI based technologies.

However, a hybrid approach would solve most of the aforementioned problems; proper research in further developing a modification of the proposed framework and the development of advanced tools to support the framework would help in realizing the use of such a framework. The biggest factors that would support the proposed and other similar frameworks would be the process of standardizing and reducing unpredictability in the way software is implemented. If the challenges one would have to meet in implementing a design is properly predicted, components which are very similar in their functionality as assumed in the design process, would be built. Building components which have predictable behavior would improve the practicality of such a framework.

Technologies such as intelligent agents, which are already in existence, as mentioned earlier, embed the intelligence inside the components themselves. Such an approach is comparable to the proposed framework and results in an automated web services environment where none of the supporting layers (inference engine or interfacing application) are required.

Other approaches can be studied and the existing methodologies can be suitably modified to focus on improving the traceability in software development. Of the processes that are being practiced, RUP achieves the desired results in complex development environments and has been a tried and tested methodology. Present processes can be further improved to map requirements and implementations to a greater degree. Better development practices can also help in improving the traceability in software development. But the framework or standard processes are the desired way of solving the problems as practices have the unpredictability of human behavior associated with it.

LIST OF REFERENCES

[1] C. Emig, J. Weiser, and S. Abeck, "Development of SOA-based Software Systems – an Evolutionary Approach," in *Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services*, 19-25 Feb. 2006, pp.182-192.

[2] M. Korsa, R. Olesen, and O. Vinter.(2002, Feb.). Iterative Software Development- A Practical View. DataTekniskForum. Horsholm, Denmark. [Online]. Available: http://inet.uni2.dk/~vinter/df-16a.pdf

[3] CASEMaker Inc. (2000, Jan.). What is Rapid Application Development. Santa Clara, CA. [Online]. Available: http://www.casemaker.com/download/products/totem/rad_wp.pdf

[4] Rational Software Corporation. (1998). Rational Unified Process: Best Practices for Software Development Teams. Cupetino, CA. [Online]. Available: http://www.augustana.ab.ca/~mohrj/courses/2000.winter/csc220/papers/rup_best_practices/rup_bestpractices.pdf

[5] J. Foreman. (1997, Jan.). Cleanroom Software Engineering – Software Technology Roadmap. SEI, Carnegie Mellon University, Pittsburg, PA. [Online]. Available: http://www.sei.cmu.edu/str/descriptions/cleanroom_body.html

[6] Q. H. Mahmoud. (2005, Apr.). Service-Oriented Architecture and Web Services: The Road to Enterprise Application Integration (EAI). Sun Microsystems Inc., CA. [Online]. Available: http://java.sun.com/developer/technicalArticles/WebServices/soa/

[7] MomentumSI. (2006). SODA-Service Oriented Development of Applications. Austin, TX. [Online]. Available: http://www.serviceoriented.org/soda.html

[8] M. M. Tanik and R. T. Yeh, "Rapid Prototyping in Software Development," *IEEE Computer*, vol. 22, No. 5, pp. 9-11, May 1989.

[9] N. Gold, C. Knight, A. Mohan, and M. Munro. (2004, Mar.). Understanding Service-Oriented Software. *IEEE Software.* [Online]. *21(2)*, pp. 71-77. Available: http://doi.ieeecomputersociety.org/10.1109/MS.2004.1270766

[10] T. Berners-Lee, J. Hendler, and O. Lassila. (2001, May). The Semantic Web. ScientificAmerican.com. [Online]. May-2001 Issue. Available:

http://www.sciam.com/article.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21

[11] G. Klyne and J. J.Carroll. (2004, Feb.). Resource Definition Framework: Concepts and Abstract Syntax. W3C. [Online]. Available: http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/#section-concepts

[12] J. J. Carroll, I. Dickinson, et al. (2003, Dec.). Jena: Implementing the Semantic Web Recommendations. HP Laboratories, Bristol, U.K. [Online]. Available:http://www.hpl.hp.com/techreports/2003/HPL-2003-146.pdf

[13] Semantic Technology Conference. (2006). Semantic Technology Primer. Wilshire Conferences Inc. [Online]. Available: http://www.semantic-conference.com/primer.html

[14] E. J. Friedman-Hill. (2006, Sep.). Jess, Version 7.0RC1. Sandia National Laboratories, CA. [Online]. Available: http://www.jessrules.com/jess/docs/70/index.html

[15] C. LeVasseur. (2001, Jul.). IT Spending: Its History and Future. Gartner Inc. [Online].Available:http://www.gartner.com/4_decision_tools/measurement/measure_it_articles/july01/mit_spending_history1.html

[16] NCSA. (2006, Jul.). MyProxy:Credential Management Service. University of Illinois, Urbana-Champaign, IL. [Online]. Available: http://grid.ncsa.uiuc.edu/myproxy/

[17] D. Felton. (2006, Aug.). Zend Framework: Zend_Authentication Component Proposal. Zend Technologies Inc. [Online]. Available: http://framework.zend.com/wiki/display/ZFPROP/Zend+Authentication+Component+Proposal+-+Darby+Felton

[18] Google SOAP Search APIs.(2006). Google Inc.[Online]. Available: http://code.google.com/apis/soapsearch/index.html

[19] Six Sigma Questions. (2000-2006). iSixSigma LLC. [Online]. Available: http://www.isixsigma.com/library/content/c010204a.asp

[20] Capability Maturity Model for Software (2006, Jan.). SEI, Carnegie Mellon University. [Online]. Available: http://www.sei.cmu.edu/cmm/

[21] J. Hendler. (2001, Apr.). Agents and the Semantic Web. Department of Computer Science, University of Maryland, College Park, MD. [Online]. Available: http://www.cs.umd.edu/users/hendler/AgentWeb.html

APPENDIX A

AUTHENTICATION USING CERTIFICATES AND PROXIES

A public key infrastructure (PKI) is a framework for trusted third-party vouching for user identities. It allows the binding of public keys to users as another mode of associating identity to user. The PKI framework is usually coordinated by software at a central location together with other interacting software at distributed locations. The public keys are typically in certificates.

The term is used to mean the certificate authority (CA) and related arrangements as well as the use of public key algorithms in electronic communications. The latter sense is erroneous since PKI methods are not required to use public key algorithms.

PKI arrangements enable users to be authenticated to each other and to use the information in identity certificates (i.e., each other's public keys) to encrypt and decrypt messages exchange. In general, a PKI consists of client software, server software such as a CA, hardware, and operational procedures. A user would have his private key (as the term suggest, this key would be confidential to the user) and would digitally sign messages using this key; another user can check that signature (using the public key contained in that user's certificate issued by a CA within the PKI). This enables two (or more) communicating parties to establish confidentiality, message integrity and user authentication without having to exchange any secret information in advance.

Most enterprise-scale PKI systems rely on networks of certificates to establish a party's identity, as a certificate may have been issued by a CA whose authority is

established for such purposes by a certificate issued by a higher-level CA, and so on. This produces a certificate hierarchy composed of, at a minimum, several groups, often more than one organization, and often assorted interoperating software packages from several sources. Standards are critical to PKI operation, and public standards are critical to PKIs intended for extensive operation. Enterprise PKI systems are often closely tied to an enterprise's directory scheme, in which each employee's public key is often stored (embedded in a certificate), together with other personal details The present leading directory technology is LDAP and, in fact, the most common certificate format (X.509) stems from its use in LDAP's predecessor, the X.500 directory schema.

The problem of assuring correctness of match between data and entity when the data are presented to the CA, and when the credentials of the person asking for a certificate is likewise presented, is difficult, which is why commercial CAs often use a combination of authentication techniques. In many enterprise systems, local forms of authentication such as Kerberos can be used to obtain a certificate which can in turn be used by external relying parties.

Proxy repository management is best exemplified best by MyProxy, an open source software for managing X.509 Public Key Infrastructure (PKI) security credentials (certificates and private keys).

MyProxy combines an online credential repository with an online certificate authority to allow users to securely obtain credentials when and where needed. Users run myproxy-logon to authenticate and obtain credentials, including trusted CA certificates and Certificate Revocation Lists (CRL). Storing credentials in a MyProxy repository allows users to easily obtain RFC 3820 proxy credentials, without worrying about

managing private key and certificate files. They can use MyProxy to delegate credentials to services acting on their behalf (e.g., a grid portal) by storing credentials in the MyProxy repository and sending the MyProxy password to the service. They can also allow trusted servers to renew their proxy credentials using MyProxy, so, for example, long-running jobs don't fail because of expired credentials. A professionally managed MyProxy server can provide a more secure storage location for private keys than typical end-user systems. MyProxy can be configured to encrypt all private keys in the repository with user-chosen passwords, with server-enforced policies for password quality. By using a proxy credential delegation protocol, MyProxy allows users to obtain proxy credentials when needed without ever transferring private keys over the network.

For users that do not already posses PKI credentials, the MyProxy CA provides a convenient method for obtaining them. The MyProxy CA issues short-lived session credentials to authenticated users. The repository and CA functionality can be combined in one service or can be used separately.

MyProxy provides a set of flexible authentication and authorization mechanisms for controlling access to credentials. Server-wide policies allow the MyProxy administrator to control how credentials may be used. Per-credential policies provide additional controls for credential owners. MyProxy supports multiple authentication mechanisms, including password, certificate, Kerberos, Pubcookie, PAM, LDAP, SASL, and one-time passwords (OTP).

APPENDIX B

FRAMEWORK PSEUDO CODE

1. RDF WRITER JAVA CLASS

```
/**
 * File: RdfWriter.java
 */

/**
 * RdfWriter - Converting a model to serialization in rdf/xml
 *
 * Date: August 30, 2006
 *
 * @version 0.1
 * @author Sambit Patnaik
 *
 *
 */


import java.io.PrintStream;

import java.util.ArrayList;
import java.util.List;
import org.apache.commons.lang.StringEscapeUtils;


public class RdfWriter {

  /** Default prefix namespace */
  public static final String NAMESPACE_PREFIX_DEFAULT = "sweto";


  /**
   * Prints the header information for the RDF file
   *
   * @author Sambit Patnaik
   * @version 0.1, created
   *
```

```
   * @param  printStream     the stream to print to
   * @param  namespacesList  List of namespaces and abbreviations
   *             array[0] = namespace
   *             array[1] = abbreviation
   */
  public static void printHeader( PrintStream printStream, List namespacesList ) {

    //Create the header string
    StringBuffer rdfString = new StringBuffer();
    rdfString.append( "<?xml version='1.0' encoding='UTF-8'?>" );
    rdfString.append( "\n<!DOCTYPE rdf:RDF [" );
    rdfString.append( "\n\t<!ENTITY rdf '" + IURI.RDF + "'>" );
    rdfString.append( "\n\t<!ENTITY rdfs '" + IURI.RDFS + "'>" );

    String[] currNamespace = null;
    String[] xmlBase = null;

    for( int i = 0; i < namespacesList.size(); i++ ) {
      currNamespace = namespacesList.get( i );
      rdfString.append( "\n\t<!ENTITY " + currNamespace[ 1 ] + " '" + currNamespace[ 0 ]
+ "'>" );
      if( i == 0 ) {
        xmlBase = new String[]{ currNamespace[ 1 ], currNamespace[ 0 ] };
      }; // if
    }; // for
    rdfString.append( "\n\t]>" );
    rdfString.append( "\n<rdf:RDF xmlns:rdf=\"&rdf;\"" );
    rdfString.append( "\n\txmlns:rdfs=\"&rdfs;\"" );
    for( int i = 0; i < namespacesList.size(); i++ ) {
      currNamespace = namespacesList.get( i );
      rdfString.append( "\n\txmlns:" + currNamespace[ 1 ] + "=\"&" + currNamespace[ 1 ]
+ ";\"" );
    }; // for

    // add xml-base
    if( xmlBase != null ) {
      rdfString.append( "\n\txml:base=\"&" + xmlBase[ 0 ] + ";\"" );
    }; // if

    rdfString.append( "\n>" );
    rdfString.append( "\n\n" );

    printStream.print( rdfString.toString() );
  }; // printHeader
```

```
 /**
  * Prints the footer for the RDF file
  *
  * @author  Sambit Patnaik
  *
  * @param out the stream to print to
  */
 public static void printFooter( PrintStream out ) {
   out.print( "\n</rdf:RDF>" );
 }; // printFooter


 /**
  * Gets the abbrev:type version of a class or property
  *
  * @author  Sambit Patnaik
  *
  * @param uri     the uri of the instance
  * @param namespaces  the List of namespaces
  *             array[0] = namespace
  *             array[1] = abbreviation
  * @return        the abbrev:type form of the type of this  class or property
  */
 public static String getShortType( String uri, List<String[]> namespaces ) {
   String abbrev = getAbbrevForURI( uri, namespaces );
   int hashIdx = uri.indexOf( "#" );
   if( hashIdx < 0 ) {
     hashIdx = uri.lastIndexOf( "/" );
   }; // if
   String name = uri.substring( hashIdx + 1 );

   return abbrev + ":" + name;
 }; // getShortType


 /**
  * Prints the classes to the writer
  * @param  ontologyModel  the ontology model
  * @param  printStream  the print stream
  */
 public static void printClasses( IOntologyModel ontologyModel, PrintStream
printStream ) {

   ILiteral literal = null;
   String literalValue = null;
   String uri = null;
```

```
    String propertyUri = null;

    for( ISchemaClass schemaClass : ontologyModel.getSchemaClasses() ) {
      printStream.println( "<rdfs:Class rdf:about=\"#" + schemaClass + "\">" );
      // output literals
      for( ILiteralStatement literalStatement : schemaClass.getLiterals() ) {
        literal = literalStatement.getObject();
        literalValue = literal.getValue();
        propertyUri = literalStatement.getPredicate().getURI();
        if( IURI.RDFS_LABEL.equals( propertyUri ) ) {
          propertyUri = "rdfs:label";
          literalValue = RdfWriter.getImprovedLabel( literalValue);
        }; // if
        printStream.println( "  <" + propertyUri + ">" + StringEscapeUtils.escapeXml(
literalValue ) + "</" + propertyUri + ">" );
      }; // for
      // whenever no literal was found, create one using the class-name
      if( literalValue == null ) {
        printStream.println( "  <rdfs:label>" + RdfWriter.getImprovedLabel(
schemaClass.getURI() ) + "</rdfs:label>" );
      }; // if
      // output parent classes
      for( ISchemaClass parentSchemaClass : schemaClass.getParents() ) {
        uri = parentSchemaClass.getURI();
        printStream.println( "  <rdfs:subClassOf rdf:resource=\"#" + uri + "\"/>" );
      }; // for

      printStream.println( "</rdfs:Class>" );
      printStream.println();
    }; // for

  }; // printClasses


  /**
   * Prints the properties
   * @param  ontologyModel  the ontology model
   * @param  printStream  the printsStream
   * @param  namespaces  the namespaces
   */
  public static void printProperties( IOntologyModel ontologyModel,
      PrintStream printStream, List<String[]> namespaces ) {

    ILiteral literal = null;
    String literalValue = null;
    String uri = null;
```

```
    String propertyUri = null;

    for( ISchemaProperty schemaProperty : ontologyModel.getSchemaProperties() ) {
      if( IURI.RDFS_LABEL.equals( schemaProperty.getURI() ) == false ) {
        printStream.println( "<rdf:Property rdf:about=\"#" + schemaProperty + "\">" );
        for( ILiteralStatement literalStatement : schemaProperty.getLiterals() ) {
          literal = literalStatement.getObject();
          literalValue = literal.getValue();
          propertyUri = literalStatement.getPredicate().getURI();
          if( IURI.RDFS_LABEL.equals( propertyUri ) ) {
            propertyUri = "rdfs:label";
          }; // if
          printStream.println( "  <" + propertyUri + ">" + StringEscapeUtils.escapeXml(
literalValue ) + "</" + propertyUri + ">" );
        }; // for
        // whenever no literal was found, create one using the class-name
        if( literalValue == null ) {
          printStream.println( "  <rdfs:label>" + RdfWriter.getImprovedLabel(
schemaProperty.getURI() ) + "</rdfs:label>" );
        }; // if
        // output subproperties
        for( ISchemaProperty parentSchemaProperty : schemaProperty.getParents() ) {
          uri = parentSchemaProperty.getURI();
          printStream.println( "  <rdfs:subPropertyOf rdf:resource=\"#" + uri + "\" />" );
        }; // for
        // output domain
        for( ISchemaClass schemaClass : schemaProperty.getDomain() ) {
          uri = schemaClass.getURI();
          printStream.println( "  <rdfs:domain rdf:resource=\"#" + uri + "\" />" );
        }; // for
        // output range
        String abbreviation = null;
        for( IRange range : schemaProperty.getRange() ) {
          uri = range.toString();
          abbreviation = RdfWriter.getAbbrevForURI( uri, namespaces );
          if( abbreviation.equals( uri ) == false && uri.indexOf( "#" ) != -1 ) {
            uri = "&" + abbreviation + ";" + uri.substring( uri.indexOf( "#" ) + 1 );
          }
          else {
            uri = "#" + uri;
          }; // if
          printStream.println( "  <rdfs:range rdf:resource=\"" + uri + "\" />" );
        }; // for
        printStream.println( "</rdf:Property>" );
        printStream.println();
      }; // if
```

```
   }; // for
  }; // printProperties

}; // class RdfWriter
```

2. ACTION JAVA CLASS

```
package framework.controller.action;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionMessage;
import org.apache.struts.action.ActionMessages;
import framework.inference.*;
import framework.util.*;


public class SearchAction extends BaseAction {

   public ActionForward executeSubmit(ActionMapping mapping, BaseActionForm
form, HttpServletRequest request, HttpServletResponse response){

      LogUtils.TRACE("-->Enter method executeSubmit on class
SearchAction");
            HttpSession session = request.getSession();

            String [] searchParams = session.getAttribute("seachParams");
            String [] searchType = session.getAttribute("stype");
            String target= session.getAttribute("starget");

            if (//serach params are not null)
                  //          Construct the XML with the search Parameters
                  //          Call the appropriate method in inference layer and pass the
XML
            forward = mapping.findForward(FORWARD_NEXT);
         }
      }
```

```
        LogUtils.TRACE("<-- Exit method executeSubmit on class
EnterSerialNumberAction");
        return forward;
    }



public ActionForward executeBack(ActionMapping mapping, BaseActionForm
form, HttpServletRequest request, HttpServletResponse response) {
        LogUtils.TRACE("<-- Enter method executeBack on class
SearchAction");
        HttpSession session = request.getSession();
        SessionGlobalData sessionGlobalData =
SessionManager.getSessionGlobalData(session);

              //if new search
              if (sessionGlobalData.getCustomerTypeFlow().equals("Existing")){
               forward = mapping.findForward(BACK_NEW_SEARCH);
        }
        else{
               forward = mapping.findForward(SEARCH_REFRESH);
        }
        LogUtils.TRACE("<-- Exit method executeBack on class
SearchAction");

        return forward;
    }

    public ActionForward executeOther(
                    ActionMapping mapping,
                    BaseActionForm form,
                    HttpServletRequest request,
                    HttpServletResponse response)throws ClientException{
        HttpSession session = request.getSession();

              //if not authenticated
              if (sessionGlobalData.getCustomerId().equals("")){

              forward = mapping.findForward(LOGIN);

              }

        return mapping.findForward(SEARCH_RESET);
        }
}
```