TUHH
Technische Universität Hamburg-Harburg

STS Softwaresysteme

Generation of Web Service Descriptions and Web Service Module Implementation for Concept-oriented Content Management Systems

submitted for the thesis defense
to attain the Bachelor Degree of
Information Technologies
by

Patrick Un

supervised by
Prof. Dr. Joachim W. Schmidt
Dr. Hans-Werner Sehring

Hamburg University of Science and Technology
Software Systems Institute (STS)

Abstract

Nowadays web services belong to the promising technologies that facilitate the communication and interaction between heterogeneous computing platforms. When it comes to providing a communication facility for a concept-oriented content management system with another system on a different computing platform, there is a need to define a communication interface and a set of communication protocols that are used by both systems in order to initiate the communication. A web services endpoint communication interface utilizes the standardized SOAP protocol and XML messages to exchange information between web services participants. Web services have many advantages such as the adoption of a standardized communication protocol, an open standard of message encoding, and ensuring low-coupling between the service provider and the service consumer. Consequently, there is the motivation of providing a web services module to the concept-oriented content management systems. This work surveys web services provision for concept-oriented content management systems as well as describes some of the design and implementation issues during the realization of the web services module.

Declaration

I declare that:
this work has been prepared by myself,
all literal or content based quotations are clearly pointed out,
and no other sources or aids than the declared ones have been used.

Hamburg, submitted on: 14th, November 2006
Patrick Wai Un

Acknowledgement

I would like to thank Prof. Joachim W. Schmidt of STS for supervising this thesis. I would like also to thank Dr. Hans-Werner Sehring and Sebastian Bossung for providing helpful guidance and experienced advices for my thesis. In chapter two, some of the illustrations are reproduced from the doctoral thesis work of Dr. Hans-Werner Sehring, used with his kind permission. Credits of the illustrations are attributed to Dr. Hans-Werner Sehring who will retain all the rights on the illustrations

dedicated to my parents, for their endless love.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.1.1 Communication beyond Content Management Systems . . . 2
1.1.2 Rationale for Web Services Adoption . . . 2
1.2 Server Module Design Objectives . . . 4
1.2.1 General Web Services Provision . . . 4
1.2.2 Server Modules as Web Services Endpoints . . . 4
1.3 Structure of the Thesis . . . 5

2 Concept-oriented Content Management **7**
2.1 Concept-oriented Content Management Systems — CCMSs . . . 7
2.1.1 Assets — the Building Blocks of CCMSs . . . 8
2.1.2 Definition of Assets . . . 9
2.1.3 Manipulation of Asset Instances . . . 10
2.1.4 Querying Asset Instances . . . 11
2.2 The CCMS Compiler Framework . . . 13
2.2.1 Overview of the Asset Language Compiler . . . 13
2.2.2 Compiler Frontend . . . 13
2.2.3 Compiler Backend . . . 13
2.2.4 Dynamic System Creation . . . 14
2.3 CCMS Component Architecture . . . 15
2.3.1 Overview of the CCMS Component Architecture . . . 15
2.3.2 Component Implementation of CCMSs . . . 15
2.3.3 Modularization . . . 16
2.3.4 Configurability of Modules . . . 16
2.3.5 Epilogue on Web Services Server Module . . . 18

3 Web Services **21**
3.1 Fundamentals of Web Services . . . 21
3.1.1 Defining Web Services . . . 21
3.1.2 Benefits of Web Services . . . 28
3.1.3 Development and Challenges . . . 28
3.1.4 Web Services Protocol . . . 33
3.2 Web Services Server Module Endpoint Requirements . . . 36
3.2.1 Generation of Web Services Description . . . 37
3.2.2 Functional Requirements of a Server Module . . . 38
3.2.3 Functional Requirements of Web Technologies . . . 38
3.2.4 Non-functional Requirements . . . 40
3.3 Web Services Technologies Survey . . . 41
3.3.1 Java based Web Services Technologies . . . 41

4 WSDL Generator and Web Services Modules Design 43
4.1 Web Services Interface Description — WSDL . . . 43
4.1.1 Mapping the Module Interface to WSDL . . . 46
4.1.2 Modeling Generic Asset Types with XML Schema . . . 47
4.1.3 Generation of XML Schema Definitions . . . 54
4.2 WSDL Generator . . . 55
4.2.1 Design Overview . . . 55
4.2.2 Generator Software Architecture . . . 57
4.2.3 Internal Design . . . 57
4.3 Web Services Endpoint Design . . . 59
4.3.1 Architectural Overview . . . 59
4.3.2 The AXIS Framework . . . 61
4.3.3 Web Services Endpoint Design Approaches . . . 64
4.3.4 HTTP Server and Embedded Web Container . . . 67

5 Implementation of the WSDL Generator and Web Services Module 71
5.1 WSDL Generator . . . 71
5.1.1 Generator Classes . . . 71
5.1.2 Methods Implementations . . . 73
5.1.3 Configuration of the WSDL Generator . . . 73
5.2 AXIS Framework Integration . . . 75
5.2.1 Setup and Configuration . . . 75
5.2.2 Generation of Service Implementations from WSDL . . . 78
5.2.3 Service Deployment in J2EE Servlet Container . . . 78
5.3 Server Module . . . 78
5.3.1 Server Module Configuration . . . 78
5.3.2 Configuration of Embedded Servlet Container — Jetty . . . 79
5.3.3 Module Initialization and Modification . . . 79

6 Summary and Outlook 83
6.1 Conclusion and Assessments . . . 83
6.2 Outlook . . . 84

A XML Schema of Generic Types 89

B XML Schema Generator Configurations 94

C WSDL Generator Configuration 96

D WSDL Document of Server Module 98

E Code Excerpt 105

F Class Diagrams 109

List of Figures

1.1 CCMS module types . . . 3

2.1 Asset entity modeling in concept-content pair . . . 8
2.2 States of asset lifecycle and state transitions . . . 14
2.3 Components configuration of the CCMS component architecture . . . 18
2.4 Component implementation with modules . . . 19
2.5 Generic object and module interface of asset modules . . . 19

3.1 Service oriented architecture triangle . . . 23
3.2 SOA triangle with service bus . . . 24
3.3 Structural parts of a WSDL document . . . 25
3.4 The abstract SOA stack overview . . . 27
3.5 The SOA stack implementing the web services architecture . . . 31
3.6 Structure of a SOAP message . . . 34
3.7 Use case analysis . . . 37
3.8 JSE web service deployment using servlet delegation . . . 39
3.9 Web services endpoint deployed with a stateless EJB component . . . 40

4.1 Asset type inheritance . . . 55
4.2 Class diagram of the WSDL generator and the symbol table . . . 56
4.3 Generators dependency among one another . . . 58
4.4 Class diagram of the strategy pattern . . . 58
4.5 Class diagram of *WSDLAttribute* and *WSDLElement* . . . 59
4.6 Class diagram of *WSDLElement* with *WSDLAttribute* . . . 60
4.7 System architecture of AXIS . . . 62
4.8 AXIS JWS deployment . . . 63
4.9 Server module design approach 1: proprietary HTTP and SOAP component . . . 64
4.10 UML deployment diagram of approach 1 . . . 65
4.11 Server module design approach 2: externalization of communication component . . . 66
4.12 UML deployment diagram of approach 2 . . . 67
4.13 Server module design approach 3: parallel component in local context 68
4.14 UML deployment diagram of approach 3 . . . 68
4.15 Server module design approach 4: embedded servlet container . . . 69
4.16 UML deployment diagram of approach 4 . . . 69

F.1 ALD2WSDLGenerator and ALD2WSDLGeneratorSymbolTable . . . 110
F.2 WSDL generator classes Part1 . . . 111
F.3 WSDL generator classes Part2 . . . 112

List of Tables

2.1 Types of modules of CCMSs 17

3.1 Top level elements of a WSDL document 26
3.2 Web services benefits 29
3.3 Mapping of generic module elements to WSDL elements 38

4.1 Asset methods exposed in the web service interface definition 45
4.2 WSDL generator Java packages listing 61

5.1 Summary of classes of the WSDL generator 74

Chapter 1

Introduction

Modern computing platforms are diversely constructed and heterogeneously structured. The need for communication between these platforms for the purpose of exchanging information has always been the focus of computing research efforts. Web services are the cornerstone of recent distributed system design representing distributed systems with advanced design and sophisticated implementation. Web services are based on open standards such as XML and SOAP. Web services interact with other distributed applications, usually within the web-tier to exchange information in a stateless and loosely coupled manner.

Some of the advantages of the web services approach are:

1. adopting widely used open standard protocols,
2. interoperating between heterogeneous platforms,
3. loose coupling of systems,
4. integrating available legacy systems.

Ever since the inception of modern computing as an engineering discipline, computer scientists and engineers have been working on similar approaches to handle cross-platform interoperable communication, with cases of moderate success occasionally. Some of the emerged technologies which are mainly deployed in the enterprise environments are RMI mechanism of the Java enterprise platform or the CORBA broker architecture of the Object Management Group. These server side enterprise technologies have been deployed in many legacy enterprise systems; occasionally still used by the developer communities when it comes to integrating legacy systems or tightly coupled systems.

Although such forms of communication patterns have been successfully deployed in many enterprise environments, there is a need for system to exchange information in a more decoupled way. System designers have come up with web services which feature more lightweight communication components that are designed with interoperability in mind and are not solely bound to one specific computing platform.

1.1 Motivation

The need for concept-oriented content management systems – CCMSs [Seh04, SS03] to perform remote operations on other similar systems over the network has led to the development of components that can handle network communication over time. In order to perform operations over the network on remote content management systems, a *server module* can be installed as a network communication entry point to the underlying content management subsystem whose operations are exposed via the

interface of the *server module* to remote client systems. This motivation of service provision is described in section 1.1.1. Section 1.1.2 explains the rationale for adopting the web services approach in constructing the *server module* network interface.

1.1.1 Communication beyond Content Management Systems

Concept-oriented content management systems [Seh04] must exchange information within its component software structure via local messages being passed among components or via local method invocations. Moreover, the systems must also interact and handle communication with remote instances of components that reside in a different secluded runtime context; that means one content management system will need to exchange information on a broad basis with another content management system. Consequently, concept-oriented content management systems have to device the afore mentioned *server module* as a network service component to handle the communication beyond the local context.

A CCMS underlies a component software architecture in order to facilitate software reuse on the component level. Instances of the CCMS software components are called *modules*. These modules have a common interface called the *module interface* which has a fixed number of identical interface methods. These methods can be invoked by another *module* in order to perform operations on the module. Module communication takes the form of method invocation in standardized unified *module interface* in order to guarantee the best interoperability and configuration flexibility between different modules. The main idea here is to allow the modules to interact with each other solely using a general and unified *interface protocol* that will remain identical across all types of modules of the systems. There are these following types of modules:

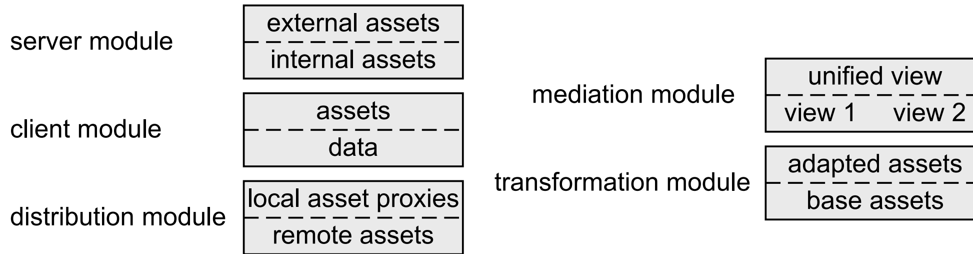
1. client module
2. transformation module
3. distribution module
4. mediation module
5. server module

These types of modules of a CCMS are illustrated in figure 1.1.

The approach of component modularization holds the advantage of flexibility in the combination of various modules in order to fulfill a certain functional requirement. By assuming a certain configuration of the desired modules; snapping and stacking the modules together regarding to a specific configuration, a hierarchy of module layers can be realized which cooperate vertically to solve certain types of problems. The layer software architecture [BMR+96] of a CCMS featuring configurable modules contributes essentially to the versatility of concept-oriented content management systems.

1.1.2 Rationale for Web Services Adoption

When it comes to the remote communication between two secluded content management systems over the network and exchanging information between interacting modules, decisions have still to be made for instance on what software component is required to perform network communication; which type of module is required to create instances of messages encapsulating the operations with corresponding parameters in order to tranmit them to the remote component. Furthermore, the *module interface* definition of a CCMS states that the unified *interface protocol* definition must be preserved on behalf of the coherence of the entire system. Therefore it would



[Seh04]

Figure 1.1: CCMS module types

be wrong if a *server module* would assume the usage of a proprietary communication protocol that is incongruent with the *module interface* protocol definition of other modules. In order to mantain and safeguard the unity of the *modules interface*, there are clear design requirements for a server module:

1. a server module must be compliant with the *module interface* protocol definition akin to other types of modules of a CCMS,
2. it must provide a standardized unified interface for remote operation invocations,
3. it must adopt standard network protocols and open technologies.
4. it must preserve the characterisitics of a layer software system, i.e. clear separation of the abstractions of system functionalities into specific software layers.
5. it must contribute to interoperability with heterogeneous systems by loosening the coupleness with other remote components over the network.

The main reason for the adoption of web services for the realization of the server module consists in the simplicity of the web services communication pattern and the interoperability with heterogeneous systems using a common web services interface.

The CRUD[1] operations which perform manipulations on a set of content artifacts of CCMS can be invoked by an equivalent module of another CCMS over the network using web services operation calls. The server module of the local CCMS proxies these operation calls on behalf of the web services clients and delegates them to the underlying CCMS base system. The remote operation calls and the corresponding arguments are mapped to simple inbound web services SOAP[2] messages and eventually delivered to the server module using a network transport protocol. The server module receives these inbound SOAP messages and relays them to an underlying, pre-configured client module of the CCMS which performs the requested operations for the web services client. Consequently, the return values or exceptions in case of an operation failure are mapped to outbound SOAP response messages or fault messages respectively.

The stateless nature of this message exchange pattern slimlines the communication system design and simplifies its implementation. SOAP messages utilize standard XML schema types to carry type information. Moreover, SOAP message contents are transmitted as XML-based structured information over the network. By adopting

[1] CRUD – denotes create, retrieve, update and delete operations that are common to both content management systems or database systems

[2] SOAP – simple object access protocol

and utilizing standard web protocols such as the HTTP protocol as the transport medium, a significant step has been taken to ensure interoperability with heterogeneous computing platforms. Consequently, a web services interface is chosen for the server module in order to fulfill the afore mentioned design requirements.

1.2 Server Module Design Objectives

As mentioned previously, the web services server module exposes the CRUD operations of a CCMS as web services operations, therefore it can extend the reach of these operations beyond the playpen of a CCMS. In the following section, a brief introduction to web services provision is given.

1.2.1 General Web Services Provision

The development of a web service can be summarized in basically two approaches:

- top-down approach,
- botton-up approach

The first approach begins with a web service interface description, in the form of the standardized web services description language – WSDL which describes the functionalities of the service endpoint. The development process continues with the implementation of the operations that are described in the service contract. The second approach reverses the process order of the first one, by implementing the operations of the service endpoint to fulfill the functional requirements of the service. The web services interface description expressed in a WSDL document is derived from the operations of the service endpoint interface. It is obtained by extracting those operations to be exported as web services operations and mapping them to the respective parts of a WSDL document. The WSDL document is then generated for both consumption by web services clients and for publishing to web services registries for archival storage purpose.

There is one main advantage to adopt the first approach than the second one: web services implementation can depend on an existing instance of WSDL document which makes the the development process more intuitive than starting the development process in the reverse order. Moreover, changes in the web services interface description can be reflected in the web services implementations as soon as possible since many development environments can generate code for stubs and skeletons for the web services endpoint implementations. These codes must only be filled in with the actual implementation details by a web services developer. Consequently the top-down development approach is adopted for developing the web services server module. By developing a generator for producing WSDL documents for CCMSs, a uniform description of the web services operations of the server module can be produced for the top-down development process.

1.2.2 Server Modules as Web Services Endpoints

The implementation of the web services endpoint can be generated afterwards by using another generator to produce the module implementation code. The task of the server module is to accept request messages from remote CCMSs acting as web services clients. These messages can be redirected to a pre-configured base module of the local CCMS on top of which the server module runs. It is required that the server module is capable of handling the parameters and return type mapping for both inbound and outbound SOAP messages. Furthermore, it must handle the request and response messages of a transport protocol layer such as the HTTP protocol.

1.3 Structure of the Thesis

This section describes the structure of the current bachelor thesis. It surveys the content of the thesis by providing brief description on the overall structure within each chapter.

Chapter 1 of this written thesis conveys the motivation behind the endeavor of providing a web service server module for a concept-oriented content management system – CCMS.

Chapter 2 elaborates on the fundamentals of concept-oriented content management systems. A detailed analysis begins with a survey of the underlying principles of CCMSs. An overview of the system architecture of a CCMS is provided for comprehension. A discussion of the core concept of a CCMS – *asset* is given in section 2.1.1, the definition of asset using *asset language* is outlined in section 2.1.2. While section 2.1.3 and section 2.1.4 discuss the CRUD operations on asset instances; there is a survey of the asset language compiler used to translate asset model definitions into configurable asset runtime systems in section 2.2. The runtime content management system underlies a component software architecture. This is described in section 2.3. An architectural overview is given in section 2.3.1. The implementation of the CCMS component software architecture is described in section 2.3.2. The asset language compiler produces generated artifacts and implementations of configurable software components called *modules*. While the module concept is further discussed in section 2.3.3, the configuration of these modules with regard to module implementation issues is described in several aspects starting from section 2.3.4 of this chapter.

Chapter 3 outlines the fundamentals of web services and its implementation on the Java platform. A fundamental description of modern web services is given in section 3.1.1. It is followed by an enumeration of the benefits of web services in section 3.1.2. The next section addresses the workflow of web services development and introduces some of the popular web services frameworks. Development challenges of web services are described in section 3.1.3. Fundamental concepts of the web services protocol is given in section 3.1.4. A detailed analysis of both functional and non-functional requirements for the web services server module is given in section 3.2 which begins with the discussion on WSDL generation in section 3.2.1. It is eventually followed by detailed discussions on the requirements in section 3.2.2, section 3.2.3 and section 3.2.4. Finally, brief discussions on issues of web services implementation on the Java Enterprise platform round up the elaboration on web services development in section 3.3 of this chapter.

Chapter 4 discusses the design issues of a WSDL document generator for the server module. In section 4.1 the design issues concerning XML schema design and methods mapping in the WSDL document are explained. This is followed by an elaboration on the design of the WSDL generator in section 4.2, giving an architectural overview of the WSDL generator. Section 4.3 describes the design issues of the web services server module. Several design approaches are shown with dicussions on the advantages and disadvantages of each design approach respectively.

Chapter 5 elaborates on the implementation of the design blueprints of the previous chapter. The software implementation for the WSDL generator is outlined in section 5.1. Section 5.2 surveys the integration of the Apache AXIS framework with the web services server module that implements a web services endpoint. Implementation of the server module is further discussed in section 5.3 which reveals important configuration details for the server module.

Chapter 6 summarizes this bachelor thesis. It sums up the design and implementation efforts by providing assessments to the degree of requirement fulfillment in section 6.1. An outlook in section 6.2 provides some suggestions and proposals on possible future developments of the CCMS web services server module regarding security, performance, and transaction processing support issues.

Chapter 2

Concept-oriented Content Management

Modern computing systems store and manipulate data that mimic and model entities of the real world. The notion of concept-oriented content management or CCM is the representation of these entities in pairwise concatenation of *content* and a *conceptual model*. This dichotomy of representation of real world entities is the fundamental principle of concept-oriented content management. The reason of using a conceptual model for content description is due to the need to structure content, for the purpose of managing and presenting it according to different presentation requirements.

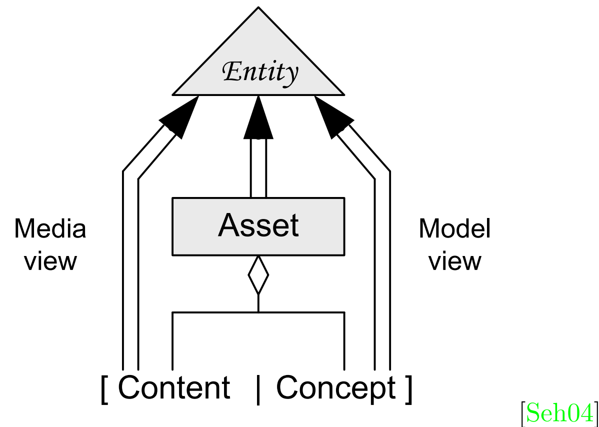
The requirements of a concept-oriented content management system demand *dynamics* and *openness* to be the underlying characteristics of the system. *Dynamics* of concept-oriented content management systems guarantees a strong coherence between a concept-oriented content model and the modeled entities. Changes in the modeled entities are reflected dynamically in the conceptual model without other technological constraints. Moreover, the definition of concepts and classification of contents can be viewed, updated and modified at any given point of time, see [Seh04]. The *openness* characteristic of concept-oriented content management systems ensures that conceptual models are open and adaptable to changes. The definition of concepts and the classification of contents should be a *non-finalizable process* according to [Cas01]. It means that the set of available concepts for modeling can be extended [Seh04]. In order to achieve the goals of dynamics and openness, concept-oriented content management systems must consist of:

- a conceptual modeling language for domain modeling,
- a model compiler which translates domain models to concept-oriented content management system components,
- a layer software architecture [BMR+96] that plugs into each other seamlessly through a predefined standardized interface to achieve interoperability between components of different types and to support system evolution.

In the following sections, different aspects of a concept-oriented content management system will be outlined and described.

2.1 Concept-oriented Content Management Systems — CCMSs

A concept-oriented content management system is a type of content management system that deals with *assets*. The management of the lifecycle of assets, for instance,



[Seh04]

Figure 2.1: Asset entity modeling in concept-content pair

the creation, retrieval, storage, modification and destruction of instances of assets, is managed by the runtime concept-oriented content management system. In the following section, the term *asset* which represents the central notion of a CCMS is explained.

2.1.1 Assets — the Building Blocks of CCMSs

According to [Seh04]:

> Assets are ontological descriptions used to classify content.

The concept of an asset is twofold. The asset representation is divided into two main parts. A concept-oriented modeling of an entity contains:

- a concept view which is also known as the model view,
- a content view which is otherwise known as the media view.

The cognition of epistemology[1] by Ernst Cassirer is treated in his main works [Cas55], it states that the observation and description of domain entities should correspond to the concrete content of that knowledge representation. Cassirer claims that the representation of domain knowledge should be a dichotomy between the abstract concept for describing the domain entity and the concrete content which represents a tangible state of that domain entity.

For the purpose of the concept-oriented content management, a CCMS harnesses this philosophy for its concept of an asset based model to mimic real world entities. The concept an *asset* of a CCMS describes and models a domain entity by:

- using a concept view for providing conceptual description of domain entities,
- using a content view for the integration and incorporation of tangible state information of the domain entities.

Cassirer proceeds with his observations in symbolic representations which have found their way into the application domain of concept-oriented content management systems. Consequently, symbolic representation of entities has become the central unit of measure – *asset* which is described by a *concept-content-pair* in a CCMS.

[1] epistemology – the theory of knowledge

Figure 2.1 depicts the concept of an asset entity which is described by the *concept-content-pair*. In the figure, the content view on the left hand side is alternatively known as the media view; whereas the conceptual view on the right hand side is also known as the model view of the asset model.

The expressiveness of asset modeling is based on the power of an *asset language*. The definition of asset is supported using this *asset language* of a CCMS. The definition relevant part of the *asset language* is called the *asset definition language* which is described in section 2.1.2 of this chapter.

2.1.2 Definition of Assets

In a CCMS assets are defined using the **Asset Definition Language – ADL**. It represents an asset definition using a schema that closely mimics the concept of a definition of a class in a conventional object-oriented programming language. It consists of both a schema-oriented language and an abstraction mechanism with the central concept of modeling called an asset *class*. For the concept-oriented modeling of assets, a *class* definition consists of the following syntactical constituents:

1. ***characteristics*** which describe different aspects of a model using an attribute-like syntax,
2. ***relationships*** which are used to maintain relational references to other asset classes,
3. ***constraints*** which model a set of rules corresponding to an asset class.

An *asset class* definition is introduced by the ***class*** keyword which is followed by the name of the asset class. An example of an asset class definition called `Picture` is given in code 2.1.1. The tangible aspects of the asset description is given by the ***characteristic*** keyword which represents the attributes with corresponding type information of the model. Relationships (either one-to-one, one-to-many or many-to-many) are captured with the ***relationship*** keyword. Inside the class definition, the ***characteristic*** keyword and ***relationship*** keyword are followed by a name of the characteristic or a name of the relationship reference respectively, then with a colon delimiting the type information afterwards. On the other hand, domain specific constraints, predicates and conditions are modeled using the ***constraint*** keyword which is followed by corresponding expressions denoting constraints or predicates. These modeling labels are used inside the class definition grouped by the ***concept*** keyword. The representation of content information is specified with the corresponding type information of the content using the ***content*** keyword. The ***content*** keyword is optional in a class definition.

A set of relevant asset classes can be grouped together to form a *model* using the ***model*** keyword of the asset language. A *model* consists of all relevant asset classes that are common to a specific modeling domain; together these relevant asset classes constitute the taxonomic description of their specific domain model.

```
class Picture {
  content contents                      : Image
  concept characteristic title          : String
          characteristic painter        : String
          characteristic creationDate   : java.util.Date
          characteristic placeOfCreation : String
}
```

Code 2.1.1: Defining an asset class `Picture`

The type system of an asset resembles the type systems of many modern programming languages, i.e. the asset type system supports the concept of a *type extension* or *inheritance*. Type inheritance is specified using the ***refines*** keyword of the asset language. Code 2.1.2 illustrates the usage of this keyword for the definition of the model `ArtHistroy` which consists of the afore mentioned `Picture` class together with a new asset class `Etching` refining the `Picture` class.

```
model ArtHistory

class Picture {
  content contents                      : Image
  concept characteristic title          : String
          characteristic painter        : String
          characteristic creationDate   : java.util.Date
          characteristic placeOfCreation : String
}

class Etching refines Picture {
  concept characteristic nameOfPlate      : String
          characteristic etchingTechnique : String
          characteristic dimension        : String
          characteristic knownPrints      : String
}
```

Code 2.1.2: Defining two asset classes `Picture` and `Etching` using the *extends* keyword to model inheritance

For the purpose of effective reuse, asset definitions can be fragmented into useful document snippets which can be imported using the ***import*** keyword into the current asset model.

2.1.3 Manipulation of Asset Instances

Instances of assets can be created, manipulated and destroyed by the **Asset Manipulation Language – AML**. In the asset language, AML is the part of the language which lends the CRUD[2] operations to the entire asset language. The asset manipulation language emplores the following keywords for performing CRUD operations on instances of assets:

1. ***let***
2. ***create***
3. ***lookfor***
4. ***modify***
5. ***delete***

Further details of the AML can be found in chapter 4 of [Seh04]. In the following description, the afore mentioned AML keywords (apart from *lookfor* which will be described in detail in section 2.1.4) are described briefly with examples:

1. the ***let*** keyword is used in general assignment of asset instances or in the construction of new asset instances in combination with the ***create*** keyword. Assuming the class definition given in code example 2.1.2, an instance of asset

[2] CRUD – create, retrieve, update and delete

`Picture` is instantiated with the ***create*** keyword and eventually assigned to the instance variable `p` of type `Picture` using the ***let*** keyword. This is illustrated in code example 2.1.3.

```
let p: Picture = create Picture {
  title := "Knabe floht seinen Hund"
  painter := "Gerard ter Borch"
}
```

Code 2.1.3: Creation of an instance of the asset `Picture` using the keyword *let*

2. the ***create*** keyword is used for the creation of new instances of assets. An example using this keyword is provided in code example 2.1.4.

3. the ***delete*** keyword is used for the reclaiming of instances of assets. Since assets are not really deleted; it states that the instances of assets which will be destroyed are only marked deleted by the system so that it is removed from the conceptual view. A CCMS registers the deletion and schedules for the reuse of the instance if necessary by management of the lifecycle of the asset instance. An example of using this keyword is shown in code 2.1.5. Here the newly created instance `p` of a `Picture` asset is deleted.

   Moreover, asset destruction can be performed iteratively on a set of asset instances that are either specified beforehand or retrieved dynamically using the ***lookfor*** operation. The operation is performed with a list semantics. In example code 2.1.6, the ***lookfor*** operation gets a list of results by using a search criteria specifying a `painter` name. The destruction of the asset instances requires that the ***delete*** operation applies the destruction iteratively on every element in the search result set. Should there be any instance of the `Picture` asset found which matches the search criteria, those instances in the result set will be deleted.

4. the ***modify*** keyword is used for the modification of asset instances such as assigning values to the *characteristic* fields, creating a new relationship with another instance of asset, etc. An example of using this keyword is shown in code 2.1.7. In this example, the newly created instance `q` is modified by assigning values to the *characteristic* fields of the asset instance, making the asset instance complete in itself according to the class definition.

2.1.4 Querying Asset Instances

The **Asset Query Language – AQL** can be used for issuing a query to search for asset instances in a CCMS using a specific search criteria as argument. The AQL can also be used in connection with specific query constraints to find instances of assets which are consequently provided as arguments to another CRUD operation to perform more complex operations on an asset instance. In terms of the asset language, the keyword ***lookfor*** is used for the asset retrieval operation. Code example 2.1.8 shows how the *lookfor* operation works. The example shows at first the creation of three instances of the `Picture` asset and one instance of the `Etching` asset.

The first query of code example 2.1.8 illustrates the retrieval of asset instances matching the search criteria of a specific painter name, this query leads to a result set containing two instances of the `Picture` asset of that specific painter. In the second query, the result set contains only one instance of the `Etching` asset. The ***let*** keyword is used to assign the result sets to the variables `resultSetA` and `resultSetB` respectively.

```
let q: Picture := create Picture {
  title := "Madame de Pompadour"
  painter := "Francois Boucher"
}
```

Code 2.1.4: Creation of an instance `q` of the asset `Picture`

```
delete p
```

Code 2.1.5: Deletion of an instance `p` of the asset `Picture` using the keyword *delete*

```
delete lookfor Picture { painter = "Eugene Delacroix" }
```

Code 2.1.6: Deletion of asset instances in a list context

```
modify q {
  title := "Die Nachtwache"
  painter := "Rembrandt van Rjin"
  creationDate : = 1642
  placeOfCreation := "Amsterdam"
}
```

Code 2.1.7: Modification of the instance `q` of the asset `Picture` using the keyword *modify*

```
let r: Picture := create Picture { title := "Marie-Louise O'Murphy"
                                   painter := "Francois Boucher" }
let s: Picture := create Picture { title := "Madame de Pompadour"
                                   painter := "Francois Boucher" }
let t: Picture := create Picture { title := "The anatomy lesson of
                                   Dr. Nicolaes Tulp"
                                   painter := "Rembrandt van Rjin" }

let u: Etching := create Etching { title := "Carceri d'invenzione"
                                   painter := "Giovanni Battista Piranesi"
                                   creationDate := "1745"
                                   placeOfCreation := "Rome"
                                   nameOfPlate := "X59B31K"
                                   etchingTechnique := "copper-acid"
                                   dimension := "50x80"
                                   knownPrints := "plate VI" }

let resultSetA : Asset* := lookfor Picture {
  painter = "Francois Boucher"
}

let resultSetB : Asset* := lookfor Etching {
  painter = "Giovanni Battista Piranesi"
}
```

Code 2.1.8: Querying asset instances using the *lookfor* operation

2.2 The CCMS Compiler Framework

In the previous section, the asset language is introduced with examples illustrating its usage. In order to harness the power of the asset language, an *asset language compiler*[3] is required to produce useful software artifacts from an initial domain model. This *asset language compiler* is described in the following sections.

2.2.1 Overview of the Asset Language Compiler

The asset language is in principle a domain-specific schema language which is used to model the domain entities of interest. It is used as input for the *asset language compiler* of a CCMS. The *asset language compiler* can be divided structurally into two main parts:

- a compiler frontend
- a compiler backend

The functionalities of a CCMS lend themselves from the entire generated set of *modules* which are produced by the corresponding generators of the asset language compiler and the combination of these *modules* to runtime software *components*.

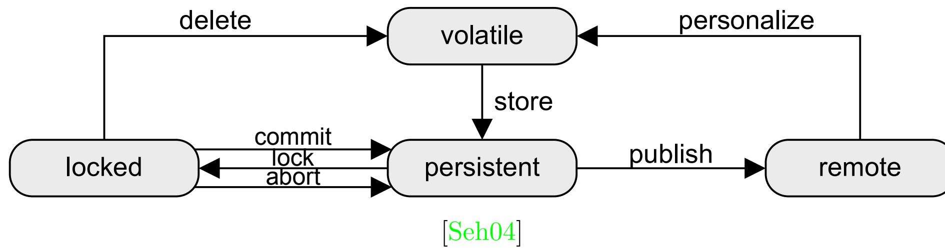
2.2.2 Compiler Frontend

The main task of the compiler frontend is to translate a user-defined domain-specific model into an *intermediate model* of internal representations which can be utilized as input by different backend generators. The compiler frontend consists of a parser and syntactic analyzer for scanning and checking asset model definitions. The compiler frontend translates a domain specific asset model into an *intermediate model* that has the appropriate form in which the backend generators can utilize as input. The *model compiler* itself and the internal representation are implemented using the Java programming language. For every asset definition of the input model, the *model compiler* creates a corresponding class in the *intermediate model* representation. The *characteristics* will be translated into a corresponding type of the implementation language which must possess this type. *Relationships* are translated into the corresponding referenced target-type which is either defined in the present input model or imported from some superior models. It is possible for the *characteristic* and the *relationship* fields of an asset class definition to possess initial values, for instance, a *characteristic* field can be initialized with a default value or a *relationship* can be created with an initial default reference binding to another asset class. These initial definitions will be translated into the corresponding *object expressions* or *asset expressions* in the model representation, consult chapter 5 of [Seh04] for more details.

2.2.3 Compiler Backend

The compiler backend consists of a number of *module generators* which are responsible for translating the *intermediate model* into specific CCMS modules that can be combined flexibly to form runtime components. The *modules* underlies a unified *module interface* which will be described in section 2.3.4 (see [Seh04] also). Because the *model compiler* utilizes the Java programming language as the target implementation language, the generated *modules* will possess the *module interfaces* defined as Java interfaces. These interfaces have their origins in the interfaces of the asset class definitions which are stored within the *intermediate model*. For the purpose of handling assets and the management of asset lifecycle in a CCMS, a set of access methods

[3] The asset language compiler is also called model compiler



[Seh04]

Figure 2.2: States of asset lifecycle and state transitions

for the possible state-changes of the asset instances are also generated from the information found in the *intermediate model* by the generators. These afore mentioned generation tasks are performed by the *API generator* of the *model compiler*.

The *API generator* generates the following interfaces:

1. *lifecycle interfaces* representing the asset instances in different states of the asset lifecycle. Figure 2.2 illustrates the possible lifecycle states of an asset instance together with the state transitions,

2. *iterator interfaces* for the set of generated objects corresponding to the generated interfaces,

3. *construction interfaces* for the fabrication and instantiation of instances of the generated types of assets,

4. *visitor interfaces* for the state transitions of asset lifecycles,

5. *visitor interfaces* for the traversal of the subtypes object structure of assets,

6. *query interfaces* for query objects,

7. *introspective interfaces* for the identification of type information using reflection.

Further details and explanations of the generated interfaces can be found in chapter 5 of [Seh04].

On the other hand, different types of concrete *modules* and *module implementations* are generated by the corresponding *module generators* of the *model compiler*. The set of *module generators* can be configured during runtime of the compiler using an XML-based configuration file. Every generator in the compiler backend is able to exchange information with other backend generators via *symbol tables*. A symbol table contains the object representation of the internal data as output of a generator; moreover, it also provides methods which can be invoked by other generators at runtime to retrieve information from the symbol table. After the creation of the afore mentioned interfaces, a specific backend generator takes the information of the *intermediate model* into account to proceed with the generation of the implementation code for those generated interfaces and the generation of the implementation code for the specific *module* itself which is composed of a set of Java implementation classes.

2.2.4 Dynamic System Creation

Several module generators can be chained occasionally to form a series of actively working backend components in order to fulfill a specific generation task. In this case, one generator consumes *symbol table* output from another preceding generator; it utilizes the symbol information in the entries of the *symbol table* of the other generator, together with the *intermediate model* produced by the frontend compiler

as input, to produce its own *symbol table* output. This processing workflow resembles the pipe and filter software architecture [SG96] in structural terms.

The main task of the API generator in the backend compiler is to generate a set of common interfaces for all the modules of a CCMS *component*. These are exported as output in an API symbol table. Further implementations of these interfaces can be generated based on the information in the API symbol table by a specific generator.

All the generated *modules* can be combined during system runtime using a specific module setup configuration that is prescribed by an XML-based configuration file. Dynamically, *modules* can be re-generated in case the model is changed. The separation of the runtime system configuration from the details of implementation of the individual *modules* lends to CCMSs a great degree of flexibility and configurability since changes in the asset models can be reflected at the level of system generation instead of at the level of the runtime system configuration and deployment.

2.3 CCMS Component Architecture

As described in the previous section, the *module generators* perform the task of generating task-specific *modules* and module implementations for a CCMS. An underlying component architecture of the system provides the foundation for the runtime system environment to these *modules*. The following sections will survey this component architecture of CCMSs.

2.3.1 Overview of the CCMS Component Architecture

The concept-oriented content management system underlies a component software architecture, see [SG96], and [Gri98]. Every component of the runtime system represents a corresponding asset which models a domain entity in a specific context. Regarding to the interpretation in [Seh04], components are responsible to represent:

- a set of assets from different domains which are grouped by a model; in such case the model is generally mapped to one component,
- a set of assets describing the same domain model but which are viewed in different context.

2.3.2 Component Implementation of CCMSs

During the generation of the runtime components by the backend generators, one *domain model* is exactly translated to one *asset model* which can include other modules from other application structures. This is shown in figure 2.3 with rectangular shapes representing the CCMS components. The lines connecting the different components in the figure represent the possible structural organization of the components among themselves. In the figure, the cooperation between the components is denoted with dotted lines connecting the cooperating components. It is obvious in the figure that flexibility in component configuration can be achieved in the vertical layers. This is depicted conceptually in the figure using a schematic structure vertically along the `Organization` axis. If one takes an example of this configuration by observing a layer structure that is formed by organizing the uppermost layer of components together with the components in the second layer from the top. The principle behind this organization can be viewed as a *personalization* relationship between the upper and the middle layers. This relationship holds in the vertical orientation when reading figure 2.3; it is observed for instance that:

$$\begin{array}{l} \mathsf{K}_1 \xleftarrow{personalizes} \mathsf{K}_{11} \text{ and } \mathsf{K}_1 \xleftarrow{personalizes} \mathsf{K}_{12} \\ \mathsf{K}_2 \xleftarrow{personalizes} \mathsf{K}_{21} \end{array} \tag{2.1}$$

The *personalization* basically denotes a variation of the component on the component that is being *personalized*, for details see [Seh04]. In the horizontal orientation along the `Application` axis, the figure illustrates the *cooperation* of the components among themselves. It can be observed in the bottommost layer in the figure that:

$$\mathsf{K}_0 \xleftarrow{\textit{cooperates with}} \mathsf{K}_1 \xrightarrow{\textit{cooperates with}} \mathsf{K}_2 \tag{2.2}$$

In CCMS terms, each *asset model* is mapped to one runtime CCMS *component* which is implemented by CCMS *modules*. The advantages of the component architecture are:

1. support of asset reuse by *component* reuse,
2. functionality reuse by *module* code reuse,
3. high level of abstraction,
4. configurability of the runtime system,
5. manageability of the lifecycle of components, and
6. separation of implementation concerns.

2.3.3 Modularization

The *modules* of concept-oriented content management systems are classified according to their respective functionality. There are mainly five types of modules:

1. server module,
2. client module,
3. mediation module,
4. transformation module, and
5. distribution module.

Each module as listed above is conceived for one particular functionality inside a component. The *modularized design* of the system enhances overall system flexibility and configurability. Table 2.1 explains the functionality of the five types of modules.

2.3.4 Configurability of Modules

The interrelationships between the different asset models are mapped to the cooperation among CCMS *components*. The cooperation among the *components* is realized as the communication between different CCMS *modules*. The configuration of the CCMS component architecture is shown in figure 2.3.

There are mainly two types of relationships between components in terms of configuration:

- *cooperation* which is a concept in the abstraction towards the application structure. In figure 2.3, the *cooperation* relationship is shown using dotted connection lines. In figure 2.4 which itself is a refinement of figure 2.3 in terms of adding an extension of details in the `implementation` orientation along the thereafter named axis as shown in figure 2.4, the *modules* which implement a specific component is shown in groups connected to the implemented component using lines heading parallel to the `implementation` axis orientation. It shows that either a component can be implemented by exactly one *module* or by several of them.

Module type	Functionality
server module	provides an interface to map between external and internal asset representations, and provides services to third party modules which communicate with the server module using a standardized protocol
client module	serves as an interface to map between the data (persistent) layer and the asset implementation system layer; it is responsible for the component persistency, i.e. managing access to databases, and managing the assets content and data
mediation module	provides a unified view for a group of base modules which have their own views themselves; it serves to glue a set of modules together which can be accessed by other modules using the unified view provided by the mediation module
transformation module	serves to adapt asset languages that conform to different asset schema definitions; it facilitates the communication between modules with different asset model definitions
distribution module	provides local proxy access to remote instances of assets

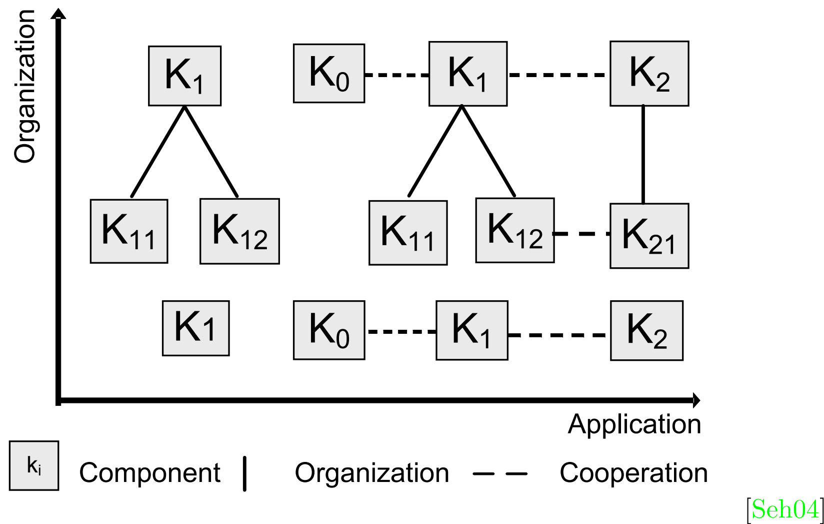
Table 2.1: Types of modules of CCMSs

Communication between *modules* is illustrated in the figure using lines in zigzag fashion. The communication spanning *modules* in the component level is defined by the *cooperation* between these *modules*. It is worth mentioning that the lattice structure of figure 2.4 shows the two distinct planes of the lattice: the front plane which denotes the *components* structure and the rear plane which depicts the *modules* structure. The *cooperation* among the *components*, for instance, (shown in the front plane) between component *C0* and *C1* or between *C1* and *C2* maps actually to *communication* between the *modules* (shown in the rear plane) which implement their respective *components*.

- *personalization* which abstracts individualization towards the organizational structure which can be divided structurally into different layers of *components* in the front plane of the lattice structure shown in figure 2.4. In this figure the *personalization* in the component plane is shown by the vertical connection lines between components distributed parallel to the organization axis orientation. In a multi-tier software architecture, such as the configuration of components shown obviously in the figure, the *modules* of a component are connected through two interfaces with other *modules* which belong to the above or underneath tiers of the configuration. These two interfaces, the *module interface* and the *object interface* (described in section 2.3.4 and 2.3.4) represent the main design goal of CCMS *modules* in order to provide a homogeneous interface of operations for the *modules*.

Different components are interchangeable in terms of the runtime configuration because the *modules* which implement the component must implement a pair of generic unified interfaces, through which all the communication between *modules* will take place in a CCMS. These interfaces are:

- the *module interface*
- the *object interface*



[Seh04]

Figure 2.3: Components configuration of the CCMS component architecture

The Module Interface

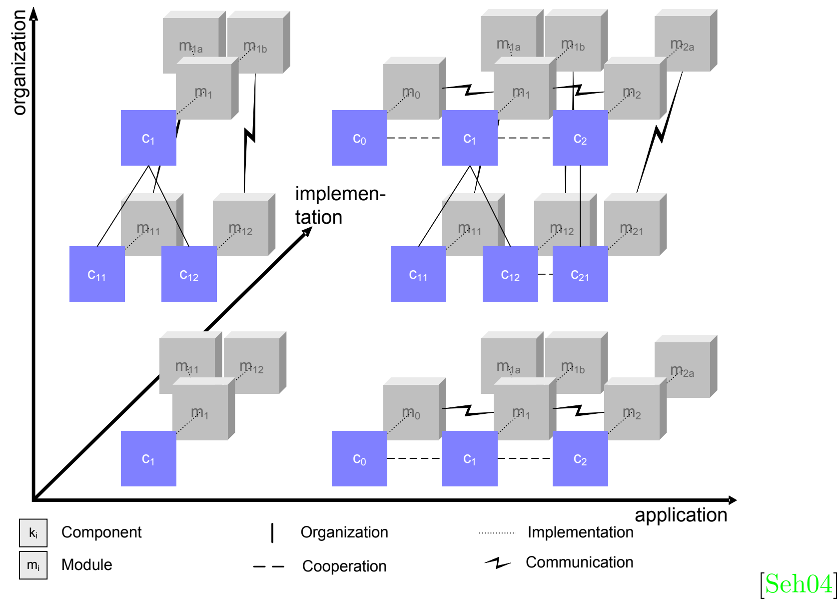
Each module must implement this interface which underlies the component it implements. The functionalities of a module derive from the concrete implementation of the *module interface*. This interface consists of mainly the CRUD operations such as *create*, *lookfor*, *modify* and *delete*. These operations take assets as arguments and return assets or the implementation of asset-relevant interfaces as return types. Consequently, the concrete implementation of the *module interface* lends the actual behaviors and functionalities to the *modules* of a component. In figure 2.5, three components of a CCMS are shown in a schematic depiction featuring individual *modules* inside each component respectively. The *module interface* of a *module* is shown using an inteface notation.

The Object Interface

In figure 2.5, the *object interface* is shown schematically using an interface notation on the left hand side of each *module*. The set of interfaces that are generated by the *API generator* is called the *object interfaces* of a module. The design of *object interface* promotes the property of *separation of concern* of the component architecture. It abstracts away the functionality concern from the structural concern of the system. The set of *object interfaces* generated includes the interfaces which are already mentioned in section 2.2.3. These interfaces must be implemented by every *module* which provides task-specific implementations for the operations of these interfaces, such that other *modules* can perform these operations on the target *module*.

2.3.5 Epilogue on Web Services Server Module

The web services server module must be compliant with the definitions of the two mentioned generic interfaces. Basically, the server module will provide web services functionality to a CCMS by introducing a web services layer on top of the CCMS subsystem. It is configured in such a manner that it must cooperate with a client module during runtime. In terms of web services provision, the operations of the server module are well-defined because the main task of the server module consists of the mapping of web services requests into CRUD operations on instances of assets



[Seh04]

Figure 2.4: Component implementation with modules

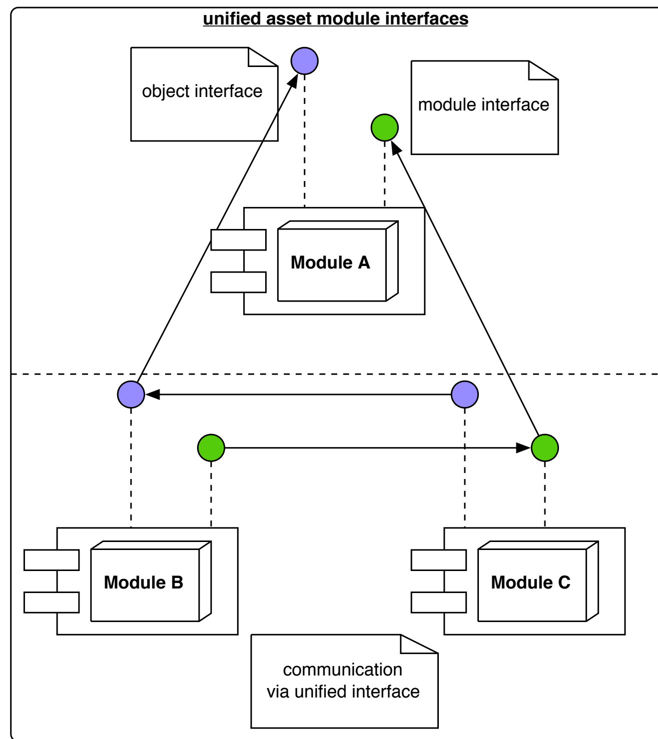


Figure 2.5: Generic object and module interface of asset modules

in a CCMS. In terms of operation semantics, the CRUD operations are perceived by the web services server module as proxiable operations performed on behalf of either another CCMS or another system which is acting as a web service client. CRUD operations are delegated to the client module on top of which the server module resides and runs; results of the CRUD operations or fault messages in case of exceptions are relayed back to the client systems by the server module which is acting as an operational deputy on behalf of the web services clients. To achieve an appropriate degree of simplicity of design, the server module should follow a message exchange pattern which is basically of a stateless and idempotent nature.

Chapter 3

Web Services

Web services operations represent a set of operations on asset instances over the network which the web services server module aspires to provide. In this chapter, the fundamental concepts of web services will be explained; together with an analysis of both the functional and non-functional requirements for the server module, a conceptual and theoretical basis will be provided for further design and implementation efforts.

3.1 Fundamentals of Web Services

In the following sections, the fundamentals of web services will be discussed in terms of the definition of a web service, the benefits of using web services, the development challenges of web services and the technological aspects of the web services protocol – SOAP.

3.1.1 Defining Web Services

The W3C[1] organization who establishes the standards for web services has defined them as follows [Con02]:

> Web services provide a standard means of interoperating between different software applications, running on a variety of platforms and/or frameworks. Web services are characterized by their great interoperability and extensibility, as well as their machine-processable descriptions thanks to the use of XML. They can be combined in a loosely coupled way in order to achieve complex operations. Programs providing simple services can interact with each other in order to deliver sophisticated added-value services.

There is also a previously used definition of web services proposed by the W3C:

> A Web service is a software application identified by a URI, whose interface and bindings are capable of being identified, described and discovered by XML artifacts and supports direct interactions with other software applications using XML based messages via Internet-based protocols.

According to IBM [dev06] web services are:

> Web services is a technology that allows applications to communicate with each other in a platform- and programming language-independent

[1] W3C – the World Wide Web Consortium

> manner. A Web service is a software interface that describes a collection of operations that can be accessed over the network through standardized XML messaging. It uses protocols based on the XML language to describe an operation to execute or data to exchange with another Web service. A group of Web services interacting together in this manner defines a particular Web service application in a Service-Oriented Architecture – SOA.
>
> Web services uses XML that can describe any and all data in a truly platform-independent manner for exchange across systems, thus moving towards loosely-coupled applications. Furthermore, Web services can function on a more abstract level that can reevaluate, modify or handle data types dynamically on demand. So, on a technical level, Web services can handle data much easier and allow software to communicate more freely.
>
> On a higher conceptual level, we can look at web services as units of work, each handling a specific functional task. One step above this, the tasks can be combined into business-oriented tasks to handle particular business operational tasks, and this in turn allows non-technical people to think of applications that can handle business issues together in a workflow of Web services applications. Thus, once the Web services are designed and built by technical people, business process architects can aggregate them into solving business level problems. Furthermore, the dynamic platform means that the engine can work together with the transmission or parts from other car manufacturers.

As an evolution of the web, web services are built on the knowledge gained from the ubiquitous distributed computing environments and technologies such as CORBA[2] and RMI[3] to enable cross platform communication and interoperability. Web services provide a standardized way for applications to expose their functionalities over the web and communication with other applications on heterogeneous computing platforms over a network, regardless of the implemention of the applications or the programming languages with which these applications are realized.

Service Oriented Architecture

The term service oriented architecture is used to indicate an architectural style that promotes software reusability by implementing web services as *reusable services*. Besides reusability it is concerned also with loose coupling and dynamic binding between services.

Conventional object-oriented architectures promotes the reusability of software by reusing classes. This approach has a fine grained nature and is suitable on a software development scope which is largely project specific. Later, component-oriented software architectures have emerged to elevate the scope of software reusability to a higher level by abstracting *software components* [SG96], [Gri98] as reusable entities which consist of a set of related common interfaces, their implementation classes, resources and configuration information.

Nowadays, the computing environments which are found in enterprises have evolved into quite complex structures due to the use of various software and hardware platforms which must communicate with each other in distributed manner. The service oriented architecture addresses these issues by using a *service* as a reusable entity. These services have typically a coarser grained nature than conventional software components. Services focus on the *functionalities* provided by their interfaces which

[2] CORBA/IIOP – Common Object Request Broker Architecture Internet InterORB Protocol
[3] RMI – Java remote procedure call

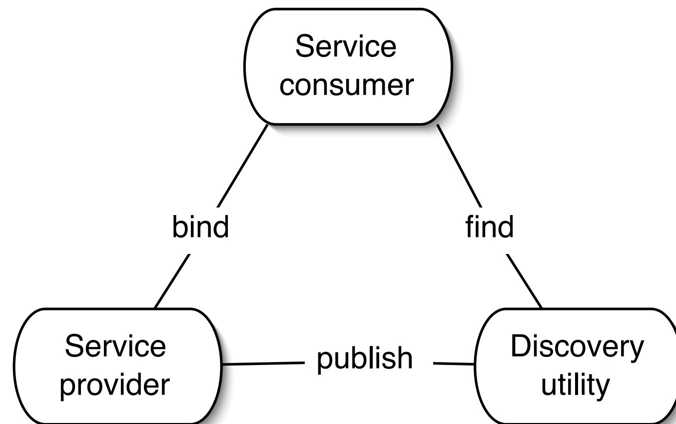


Figure 3.1: Service oriented architecture triangle

are well-defined and through which service client can communicate with the services using standardized protocols.

A service oriented architecture has three main structural constituents:

1. **Service provider** implements business logic of the service and exposes these services through its interfaces.
2. **Service consumer** discovers service from registry and accesses the web services by invoking the methods in the interfaces of the service provider. (Service consumer can be services themselves)
3. **Service registry** stores the published service descriptions by the service provider and enables service consumers to look up services.

This structure is sometimes called the SOA triangle which denotes the web services architecture as an overview, see figure 3.1.

The middleware infrastructure that is needed to enable the service provider to publish service for provision; and to enable client to access the services is referred in the literature as the *service bus* (see figure 3.2), the concept of a service bus is treated in detail in works by [MW06] and [Nl04].

Describing Web Services

The web services standards include:

- **WSDL**[4] – a description language used to define the interfaces provided by a web service, in a way that is independent of the platform on which the web service is deployed,
- **UDDI**[5] – a provision for a registry to store the service definitions.

A WSDL document describes a web service in terms of the functionalities that the web service provides and the data types that each operation requires as input parameters or output return value.

It abstracts away from the concrete implementation of the web service at the protocol level. The web service is defined in abstract terms and then mapped to one

[4] Web Services Description Language [Con01]
[5] Universal Description Discovery and Integration [ftAoSISO06a], [ftAoSISO02]

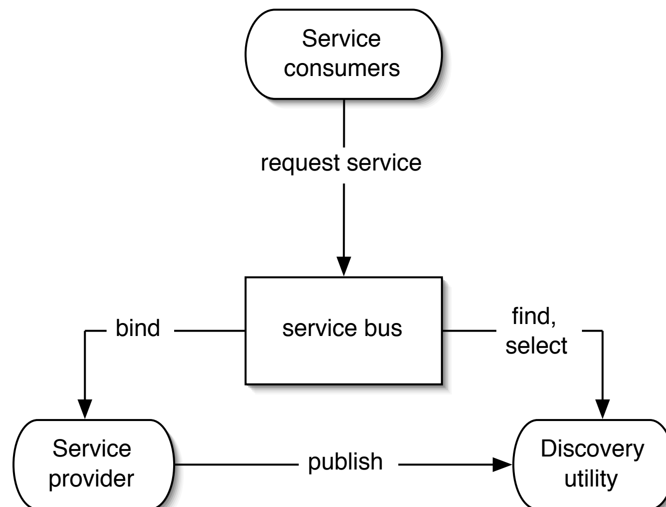


Figure 3.2: SOA triangle with service bus

or more specific protocol by using concrete protocol bindings. A binding specifies the way in which the abstract input and output messages of the web service are mapped onto a protocol. Generally used binding protocols are SOAP (see section 3.1.4). A WSDL document also contains a set of address url at which the service is deployed and can be accessed. The structure of a WSDL document is depicted in figure 3.3 on page 25. In its serialized form, the structure of a WSDL document is shown in the WSDL document code example 3.1.1 on page 25.

The top-level elements of a WSDL document are given in table 3.1.

An overview of the interface definition of a web service can generally be obtained from the information of a WSDL document. However, WSDL documents are mostly generated and consumed by software tools rather than produced from scratch. Web services clients depend on the WSDL document for critical information in order to consume the web services. For this reason, there are many tools that can help generate the client side proxy code from the WSDL document automatically. Many programming languages are now served by the different language bindings so that clients which are programmed with different language can access the web services by just retrieving the WSDL document of interest and utilizing the web service proxy code to invoke the web services.

On the server side in terms of web services implementation, there are also a wide range of tools that will start the development process by starting from the WSDL document. Server side code stubs can be generated on the fly and implementation code skeletons are also provided; these code artifacts can be easily filled in and programmed to implement the web services in an efficient manner.

Discovery of Web Services

The WSDL document produced by a web services provider can be published to a web services registry. There are two major registry standards:

- the UDDI registry[6],

[6] version 2.0 is the current standard, an upcoming version 3.0 is in the working process [ftAoSISO02]

```
<wsdl:definitions ....>
    <!-- Import definitions from external sources -->
    <wsdl:import ..../>

   <!-- Definitions of types used only in this WSDL file -->
    <wsdl:types ..../>

    <!-- Definitions of messages for this web service -->
    <wsdl:message .../>

    <!-- Definitions of the interfaces and operations -->
    <wsdl:portType .../>

    <!-- Concrete bindings of interfaces and operations to protocols -->
    <wsdl:binding ..../>

    <!-- Defines the service and supplies the protocol address -->
    <wsdl:service ..../>
</wsdl:definitions>
```

Code 3.1.1: The logical structure of a WSDL document

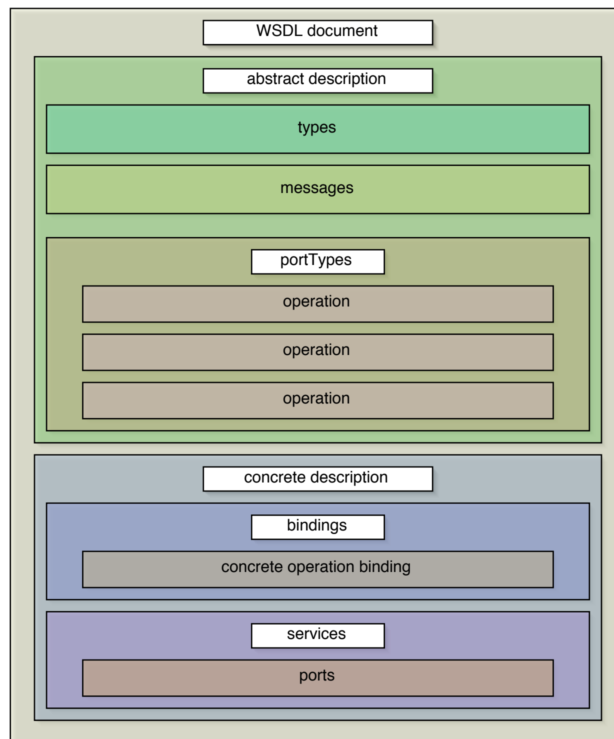


Figure 3.3: Structural parts of a WSDL document

WSDL element name	**Usage description**
import	enables web services that are defined in other WSDL documents or document fragments to be imported into the current WSDL document as required; therefore promotes document reuse. This approach is mainly used in managing large and extensive interfaces in order to allow different web services to share the same data types or to separate the definition of a web service and its protocol bindings.
types	defines the type system which is used in the web service messages exchange between the service and its client. Currently the W3C standard [Con01] advocates the use of its XML schema type system which is an XML schema language to define data types, see [Con04b].
message	describes the data that is exchanged between the web service and its clients in terms of the data types defined within the *types* element. Each message is an abstract representation of the request and response message defined that corresponds to a concrete protocol message such as SOAP.
portType	defines an abstraction for the web service endpoint interface. The functionalities of the web service are represented as abstract operations of the interface.
binding	defines the way in which the operations defined in the *portType* element and the messages defined in the *message* element are mapped to their concrete representation when a specific concrete protocol is bound as transport mechanism. If the common SOAP protocol is used, WSDL has extensions that allow the description of the header, body and the fault parts of a SOAP message.
service	contains the *port* element which contains the address url at which a concrete binding of the *portType* on which the web service is deployed. A web service client can use this piece of information to invoke the operations of a web service. It is possible to have more than one port within the WSDL document. The *service* element serves to group related ports together that will represent a web service in its entity.

Table 3.1: Top level elements of a WSDL document

- the ebXML registry[7].

The UDDI registry is intended primarily for the publication of metadata relating to web services. It allows the service provider to advertise service information that includes the location of a WSDL document and other related documentation. A registry can also contain classification information than can assist the web service clients in looking up services regarding a specific interest or category. See [Nl04] for further information on web services registries; see [AM02] and [GDS04] for a detailed treatment on the UDDI registry.

The ebXML registry standard was created by OASIS and is aimed at the E-commerce market. It is currently primarily deployed in enterprise environments. The repository allows storage of web services description documents as well as business

[7] ebXML – electronic business using extensible markup language, sometimes referred to as ebXML repository

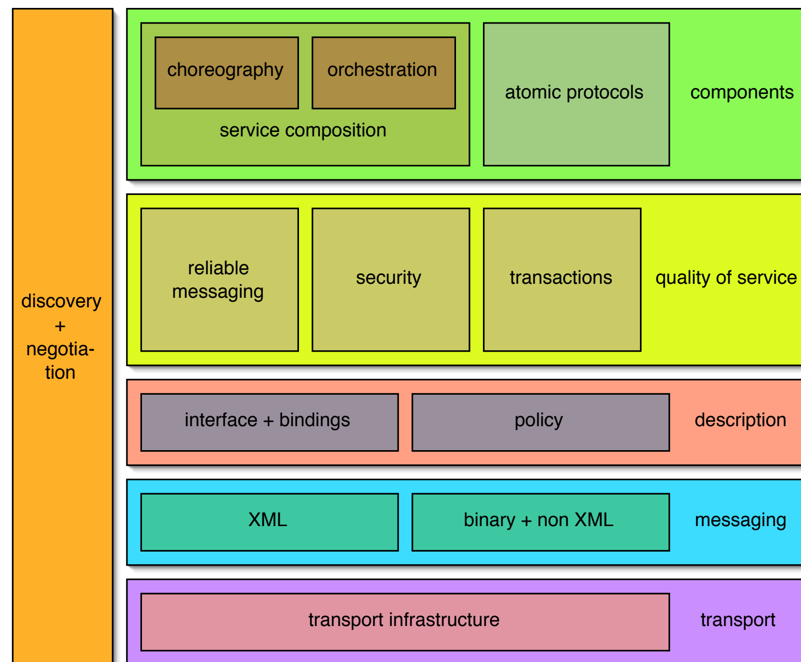


Figure 3.4: The abstract SOA stack overview

documents and other data of a business scenario; in this sense it is a different approach to management of storage of web services definitions than the UDDI registry which is mainly focused on a web services centric middleware provision. See [Wal02], [GD02] and [KWW01] for further treatment on this topic.

Frameworks for Web Services

Web services technology is a basis for implementing service oriented architecture, and the *service bus* as mentioned in section 3.1.1 is the centerpiece of the architecture implementation. The high level architectural overview of the service bus is illustrated in figure 3.4.

The figure depicts an abstract architectural overview of the service bus as a stack of functionalities of the service oriented architecture [Erl04, MW06].

The bottom layer presents its capabilities to cope with various transport protocols to communicate between a service and a requester. The messaging layer on top of the transport layer enables the bus to deal with messages.

The next layer of the bus facilitates and deals with the description of services in terms of functionalities supported, *quality of service* of these functionalities and the supporting binding mechanisms. The actual quality of the services of the bus is enforced by the appropriate parameterization via web services *policies* residing in the underneath layer. The quality of service layer copes with security aspects such as message integrity and, confidentiality and non-repudiation; moreover transport reliability of messages and transaction management support is also settled in this layer.

The top layer represents the various kinds of virtual components that web services have at their disposal. This layer encapsulates components that facilitates *atomic services* which are not composite services as compared to their composite partners juxtaposed next to them in figure 3.4. The *Composite services* are the constituent parts of the framework that supports the choreographies of web services by the *service*

bus. The coordination of participants among each other is controlled by *agreement protocol*.

The vertical layer provides features for the discovery of services and the negotiation for agreement on a mode of service interaction between requester and the service provider.

3.1.2 Benefits of Web Services

As mentioned briefly in section 1.1.2, web services have characteristics that are advantageous in terms of simplicity, efficiency, and operability, which have motivated and led to the development of a server module and the corresponding interface for the CCM system. The reason for the popularity and widespread adoption of web services underlies the following benefits:

Application development has been complicated by the requirement that a particular application support a specific type of client or even several types of them simultaneously; web services due to their interoperability with other web services, simplify this facet of the application development process.

3.1.3 Development and Challenges

The challenges faced by web service developers are numerous. Web services are the evolution of a long line of web technologies, they are far from mature. At the present stage, web services technologies are undergoing rapid evolution. Despite of the variety of development tools and implementation platforms, the challenges consist of continuous evolving technologies and standards. Consequently, developers are required to adapt to new technologies vigilantly.

Factors that will influence the development of web services in long term are:

Evolving Technologies and Tools

Many web services based solutions are still in their infancy. Web services rely on a collection of technologies, standards and specifications, though more new standards are currently being defined to enrich the potential of the web services platform. Enterprise computing systems often utilize web services as a means to distribute information, and many more systems use web services to conduct business process that requires transaction capability. Since transaction is crucial to enterprise business processes, the corresponding web services standard has just been proposed, not long before there was no universally accepted standards for conduction transactions in web services terms.

An additional challenge in the development of web services is the coordination of multiple services for processing business logic. Often a seemingly single business process is implemented as a series of stages in a real workflow, and each stage of the workflow might be implemented as a separate service. Consequently, all the services must coordinate with each other during the various processing steps. The corresponding standards are currently under reviews.

Security

Ensuring security is important for web services as for any web applications. The fact that nowadays applications on the web open up the business processes and data of an enterprise to distributed clients, has made security even a crucial factor in the design and development of web services.

When information assets are exposed in less-protected environment such as in web services, it becomes crucial to maintain security measures to preserve integrity of the

Characteristics	**Benefits**
interoperability	permits different services on distributed systems to run on a variety of software platforms and architectures. The development in this arena is due to the consolidation of cross platform systems which have functionalities added over time, the management of them which are probably implemented with different languages and running on heterogeneous platforms poses issues of integration. In order to be interoperable, web services can be emplored to interface between these systems and architectures.
wide acceptance of web technologies	due to the ubiquitousness of the web and web technologies, web services that utilize these technologies and infrastructure can leverage the advantages of this medium. On the other side, web services clients are also well served due to the wide availability of web based clients.
integration	many existing enterprise information systems have an enormous data volumn at their storage disposal; the cost to replace legacy system is probably prohibitive and thus not an option. Web services promotes integration of these legacy systems by interfacing them with a layer of loosely coupled middleware protocol, allowing system application development to commoditize exisiting information assets for reuse. Consequently consistently standard ways can be provided by using web services to access middle-tier or even data-tier services to integrate them with other applications.
open standards	Web services are extensively based on standardized protocols and open standards. There is a large set of tools, production technologies available to make web services development more efficient.
dynamic heterogeneous client support	one main goals of web services is to improve interoperability, exposing existing applications as web services enhances their reach to different clients. This is platform independent in terms of the implementation language with which the client is programmed or the architecture on which the client is running.
efficiency and productivity	application development such as web services prototyping is supported by the availability of many useful tools for web services. Thus productivity of web services development is enhanced on magnitudes that have not been reached in conventional sense since the development for other distributed computing environment have relied on a set of not-always-compatible technologies or diversed types of middleware software which are incompatible. One prominent example is CORBA which uses proprietary interface protocols for remote communication, integration with other backend systems have occassionally evolved into maintenance nightmare.

Table 3.2: Web services benefits

sensitive information from being tempered with by unauthorized clients while still providing easy web services access to other assets by web services clients.

One of the difficulties in handling distributed systems is providing an integrated *security model* that has to be compatible with existing mechanisms. In cases where web services clients have to access secure sensitive information, the security model has to ensure high security while remaining as unobtrusive and as transparent as possible. The key issues of security of web services are concerned with *authentication*, *authorization*, and ensuring *confidentiality*. Web services standards of security are crucial and therefore a highly prioritized area for the service oriented architecture. There are now many evolving working standards in progress such as *WS-Security* which ensures message level security of web services, see [ftAoSISOwssw06, dev04].

Scalability and Reliability

Developing web services often means handling large-scale distributed applications which require reliability of service and scalability for future growth.

Reliability of the web services represents how well the services can maintain their services regarding to service quality. Often reliability is measured by the number of failures occuring in a given period of time. For web services to be reliable, the infrastructure on which the web services are deployed, mostly on web application platforms using underlying HTTP, SOAP protocol nowadays, must be reliable. Sometimes reliability is however difficult to achieve because of the relatively unreliable nature of these infrastructure platforms; for instance, the HTTP protocol as transport provides only a best effort delivery and does not guarantee packet correctness or retransmissions. Web services are considered reliable if they can cope with the issues of these platforms and handle changes using configurations dynamically.

There are efforts in progress in the fields of web services reliability adopted by standard organizations.

Web services which scale effectively can handle a large number of client interactions. The web services infrastructure must efficiently manage system resources and services. One of the possible bottleneck is the effectiveness of handling of XML parsing, deserialization and serialization by the underlying web services platform. Since these processes are computing intensive if there is a large number of client connections to handle because the XML format itself is rather verbose that can increase payload size of network packets prohibitively if the size of message is getting large, the web services platform must be design with this issue in mind to avoid significant performance drawbacks.

To achieve reliability and scalability, web services must be flexibly deployed on server platforms that can scale to anticipated client volumes; moreover they must be design to be easily configurable. Some working solutions to achieve scalability in the middleware layer have been successfully deployed in many web applications such as the use of a clustering hardware environment. It is helpful to handle scalability of the systems without requiring the applications to be redesigned or reordered.

Web Services Frameworks

As mentioned in section 3.1.1 about the abstract service oriented architecture stack (SOA stack), there are numerous efforts which have been undertaken to realize the stack, the figure 3.5 on page 3.5 illustrates the implementing standards which have been adopted overtime by the standard organizations.

Transport Service

The above diagram provides an overview of the SOA stack implemention relating to the current and emerging technologies of the web services platform. Web services

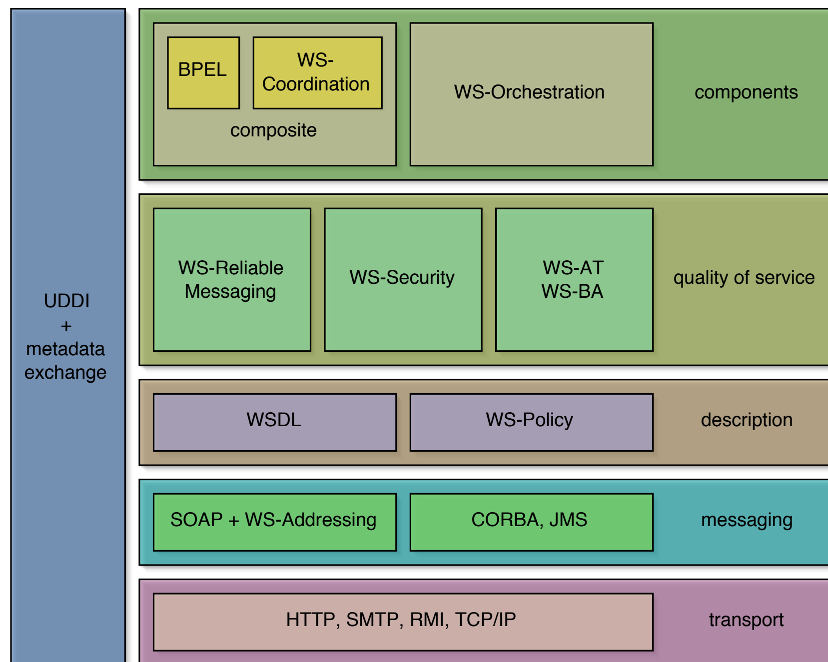


Figure 3.5: The SOA stack implementing the web services architecture

are inherently transport neutral; the fact that a variety of standard protocols are supported to transport web services messages proves that the flexibility of the services architecture. These protocols are illustrated in the transport layer of the diagram above.

Messaging Services

The messaging services component of the framework contains the most fundamental web services specifications and technologies, including XML, SOAP and WS-Addressing. Collectively, these specifications form the basis of interoperable messaging between web services. WS-Addressing provides an interoperable way of identifying message senders and receivers that are associated with message exchange. It decouples address information from the specific transport protocol used by providing a mechanism to embed target, source address information directly within the web service message. The specification defines XML elements to identify web services endpoints to secure peer communication in the exchanged messages. WS-Addressing defines two interoperable constructs: *endpoint references* and *message information headers* which can be processed indenpendently by the addressing framework. Detailed specifications can be obtained in [Con05, Nl04].

Service Description and Discovery

Above of the messaging services layer lies the service description layer. Besides the web services description language WSDL which is mentioned in section 3.1.1, the WS-Policy specification also finds its way in this layer. WS-Policy framework offers a flexible way to associate policy expressions with web services. The specification [Con06d] defines a common framework for services to annotate interface definitions to describe their service assurance qualities. This is realized in the framework as *requirements* which are machine-readable expressions containing assertions of the services. The WS-

Policy framework allows the use of algorithms to determine what concrete policies are applicable to allow requester and provider to interact. Other higher-level functionalities provided by the framework at the upper layers such as security, transactional supports and reliable messaging rely on the WS-Policy framework extensively.

Service discovery is a realm that is concerned with lookups of metadata about the web services. This task is handled by UDDI registries which is mentioned previously.

Quality of Service

Standards and specifications in this layer are related to the quality associated with interaction with web services. They specify the requirements and issues regarding overall reliability of web services, for instance the security in the interaction, reliability in the message delivery and support for transactions based on agreement and 2-phase transaction protocols.

WS-Security is the basic building block for secure web services. Modern web services on distributed middleware rely on transport-level support for security. Minimum secure communication channel is supported at this level by HTTPS and basic client authentication. WS-Security utilizes existing security infrastructure of models such as a PKI[8] of digital certificates or authentication framework like Kerberos. The specification [ftAoSISO04] defines concretely the way to use existing security models in an interoperable way for web services. Moreover, further research in the trust model has led to the development of WS-Trust [ftAoSISO06b] which defines an extensible model for setting up and verifying trust relationships.

WS-ReliableMessaging of IBM [dev05e, dev05f] defines a protocol that ensures reliable delivery of messages with specified assurance of message exchange. It specifies these assurances in terms of delivery pattern:

- *in-order delivery*,
- *at least once delivery*,
- *at most once delivery*

The above assurances can be combined to allow bridging different message oriented middleware infrastructures into a single, logical end-to-end reliable messaging model.

The support of transactional functionalities are mainly handles in the web services domain by three standard specifications:

- WS-Coordination,
- WS-Atomic Transaction and
- WS-BusinessActivity

WS-Coordination [dev05d] is a mechanism for initiating and agreeing on the outcome of multipart, multimessage web services tasks. It consists of three key elements:

- *coordination context* is associated with exchanges during the interaction of web services. It contains the addressing endpoint reference of a coordination service and information that identifies the task being coordinated,
- *coordinator service* provides a service to start and terminate a coordinated task; it allows participants to register the task and produces the coordination context within every exchanged messages between the service participants,
- *coordination interface* is used by the participants to get the outcome of the task being coordinated.

[8] PKI – a public key infrastructure

WS-AtomicTransaction [dev05b] and WS-BusinessActivity [dev05c] are two particular protocols that extend the WS-Coordination protocol to define specific ways to reach overall outcome agreement.

WS-AtomicTransaction defines a specific set of protocols that plug into WS-Coordination to implement the traditional two-phase atomic ACID transaction protocols. Transaction protocols for business transactions have to deal with long-lived activities. These differ from atomic transactions in that such activities can take much longer to complete. In this respect, mechanisms are introduced for fault and compensation handling to reverse the effects of tasks that were completed previously within a business activity, such as compensation or reconciliation.

WS-BusinessActivity defines a specific set of protocols that plug into the WS-Coordination model to provide long-running, compensation-based transaction protocols.

Service Composition and Components

The uppermost layer of the diagram depicts the composition framework for web services. Business Process Execution Language for Web services BPEL4WS [dev05a] provides a language to specify business processes and process states and how they relate to web services.

The BPEL4WS specification explains how a business process uses web services to achieve its goal, including specifying web services that a business process provides. Business processes specified in BPEL are fully executable and are portable between BPEL-conformant tools and environments. A BPEL business process interoperates with other partner web services, whether these web services are realized based on BPEL or not. Consequently, BPEL supports the specification of business protocols between partners and views on complex internal business processes.

BPEL supports the specification of a broad spectrum of business processes, from fully executable, complex business processes over more simple business protocols and constraints of web services. It provides a long-running transaction model that allows increasing consistency and reliability of web services applications. Correlation mechanisms are supported that allow identifying statefull instances of business processes based on business properties. Partners and Web services can be dynamically bound based on service references.

3.1.4 Web Services Protocol

The service bus of the service oriented architecture mentioned previously underlies a message transport mechanism. A web services protocol represents a significant underpinning of web services runtime infrastructure. It is concerned with the exchange of structured types information between web services participants. There is one mainstream web services protocol implementation in use nowadays — SOAP.

SOAP

SOAP[9] is the standard common messaging protocol used by web services. It is the de facto standard for the web services stack in general. SOAP's primary application is in the domain of B2B and enterprise application integration; being a truly effective web services protocol, SOAP is designed platform-independently, be flexible, and based on standard, ubiquitous technologies. Unlike earlier enterprise technologies, such as CORBA, SOAP enjoys widespread use in web services since its inception, and has been endorsed by most enterprise software vendors and major standards organizations such as W3C, WS-I, OASIS.

[9] SOAP – Simple Object Access Protocol

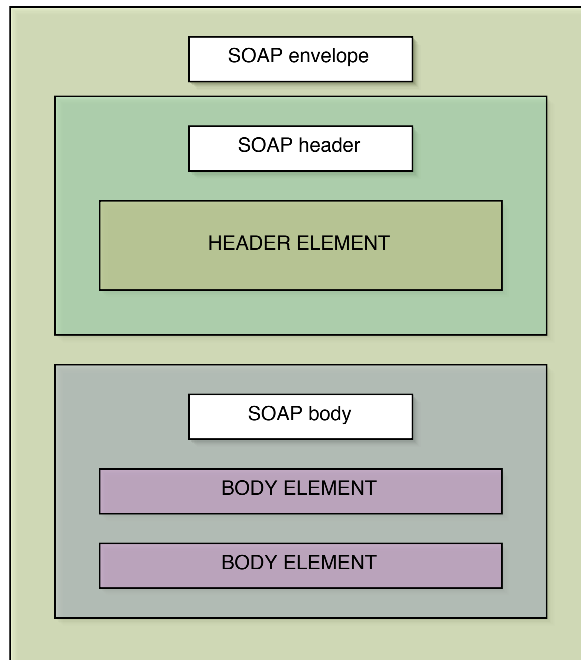


Figure 3.6: Structure of a SOAP message

The SOAP protocol is concerned with the encapsulation of web services messages, encoding them as XML data and defining the rules for transmitting and receiving that data.It is a network application protocol.

SOAP provides four main capabilities:

- a standardized message structure based on the XML Infoset,
- a processing model that describes how a service should process the messages,
- a mechanism to bind SOAP messages to different network transport protocols,
- a way to attach non-XML encoded information to SOAP messages

A SOAP XML document instance is called a SOAP message which is usually carried as the payload of some other network protocol. The most common way to exchange SOAP messages is via HTTP. SOAP messages are exchanged between applications on a network and are usually not meant for human consumption. HTTP is just a convenient way of sending and receiving SOAP messages. Other transport layer protocols, e.g. SMTP, FTP and JMS can also be used.

A SOAP message is the basic unit of communication between SOAP nodes. It consists of a SOAP envelope that contains zero or more SOAP headers. The figure 3.6 shows the structure of a SOAP message. The SOAP headers are targeted at any SOAP receiver that might be on a SOAP message path. The SOAP envelope also contains a SOAP body that contains the message payload or business information. A SOAP body might contain, for instance, a service request and input data for the service to process. While processing a SOAP message, a SOAP node might generate a fault condition. If this happens, a SOAP node returns a SOAP message containing a SOAP fault.

A SOAP node is an implementation of the processing rules described within the SOAP specification [Con03c, Con03b, Con03a] that can transmit, receive, process, or

relay a SOAP message. Although the SOAP node implements the SOAP processing model, it can also access any services that underlying network protocols might provide. It does this through a SOAP binding that specifies the rules for carrying a SOAP message on top of some other underlying network protocol.

SOAP nodes can send and receive SOAP messages. If a SOAP node transmits a message, it is called a SOAP sender; if it receives a message, it is called a SOAP receiver. Some SOAP nodes might both receive and transmit messages. In this case, they are called SOAP intermediaries. The SOAP sender that first builds the SOAP message is called the initial SOAP sender. The final destination of the message is called the ultimate SOAP receiver. This SOAP node is responsible for processing the payload of the message that is contained in the SOAP body.

The following XML instance document snippets are excerpts from SOAP messages representing a SOAP request and the corresponding SOAP response:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <soapenv:Body>
        <ns0:getClass
            xmlns:ns0="urn:de.tuhh.sts.cocoma.generic"
            soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
          <argument0 xsi:type="xsd:string">hello world</argument0>
        </ns0:getClass>
    </soapenv:Body>
</soapenv:Envelope>
```

Code 3.1.2: Logical structure of a SOAP request message (excerpt)

The request is processed by the intended SOAP receiver node which generates the following response:

SOAP specifies the overall structure of a message, together with rules for wrapping SOAP messages using the underlying transport protocol such as HTTP. There are two slightly different ways to construct a SOAP message depending on whether the message has any attachments. The SOAP version 1.1 specification requires that a message be constructed as shown in the structure illustration above. The outermost layer of the message is a protocol-specific wrapper, the nature of which is defined in the specification with HTTP as the concrete transport binding protocol. Inside this wrapper is the SOAP message itself, basically an XML instance document that conforms to the SOAP XML schema definition. It consists of an envelope, a header part, a body part, and optional additional content.

The SOAP envelope is a top-level XML element that serves as a container for the rest of the message. The SOAP header is optional, and if present, must be the first element in the envelope. It is intended to be used to carry information that can be used in the processing or routing of the message payload, such as a digital signature to secure the integrity of payload data or authentication information to validate the identity of the message sender.

The SOAP body is the only mandatory part of the envelope and contains the actual payload intended for the ultimate recipient of the message. It must either follow the SOAP header or, if the header is omitted, be the first element in the envelope. Following the body element, it is possible to include additional content, the interpretation of which, like the payload itself, is entirely dependent on the sending and receiving node.

Everything within the SOAP envelope must be encoded in XML. For the purpose

```
<soapenv:Envelope
         xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
         xmlns:xsd="http://www.w3.org/2001/XMLSchema"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <soapenv:Body>
        <ns1:getClassResponse
             xmlns:ns1="urn:de.tuhh.sts.cocoma.generic"
             soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
           <getClassReturn href="#id0"/>
        </ns1:getClassResponse>
        <multiRef
             xmlns:ns2="urn:de.tuhh.sts.cocoma.generic"
             xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
             id="id0" soapenc:root="0"
             soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
             xsi:type="ns2:AssetClass">
          <id href="#id1"/>
          <type xsi:nil="true" xsi:type="ns2:AssetClass"/>
          <name xsi:type="xsd:string">Asset</name>
          <superclass xsi:nil="true" xsi:type="ns2:AssetClass"/>
        </multiRef>
        <multiRef
             xmlns:ns3="urn:de.tuhh.sts.cocoma.generic"
             xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
             id="id1" soapenc:root="0"
             soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
             xsi:type="ns3:ID">
           <componentName xsi:type="xsd:string">
                componentname=generic
           </componentName>
           <moduleName xsi:type="xsd:string">module=CocomaSModule</moduleName>
           <internalID xsi:type="xsd:string">
               uuid=c9c5a175-60fb-426e-a2df-c2b97b95dcec_bye_bye
           </internalID>
        </multiRef>
    </soapenv:Body>
</soapenv:Envelope>
```

Code 3.1.3: Logical structure of a SOAP response message (excerpt)

of the serialization of binary data, it is not always practical to have this data encoded using XML. Consequently, an additional specification called *SOAP Messages with Attachments* [Con00] has been adopted for the specific purpose of handling SOAP messages with attachments. It has become the de facto standard for packaging SOAP messages that require non-XML encoding of the content.

3.2 Web Services Server Module Endpoint Requirements

A web services endpoint is a software implementation that is capable of performing handling of web service requests and responses. In terms of the component architecture of concept-oriented content management systems, such a software implementation is mapped to a server module. The server module is a component software that

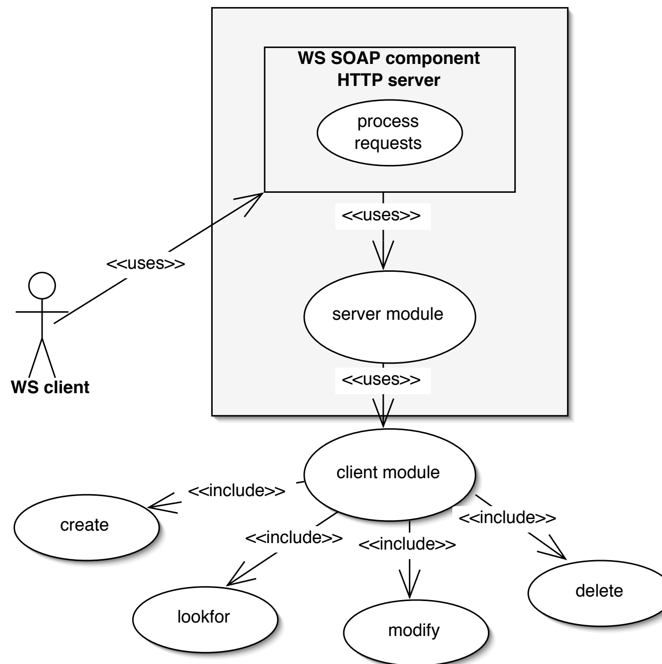


Figure 3.7: Use case analysis

adheres to the module interface definition of a CCMS and contains methods defined in the *module interface* that are common to all other types of modules.

The asset languages of CCMSs as described in the sections 2.1.2, 2.1.3 and 2.1.4 are responsible for the creation, retrieval, manipulation, and destruction of the assets. Exporting these CRUD operations to the web services endpoint is the main task of the web services server module. A use-case diagram illustrating the requirements are depicted in figure 3.7 on page 3.7.

3.2.1 Generation of Web Services Description

The structure of a WSDL document has been described in section 3.1.1. In order to translate the common asset methods of the generic module interface into web services methods for the service endpoint, these methods must be mapped to the logical structural elements of a WSDL document.

One of the main development approaches of a web services endpoint is to select whether to start from scratch code or from a WSDL document. These development issues are already described in section 1.2.1.

It is worthwhile mentioning that by adopting the top-down development approach, i.e. starting with a WSDL document provides the benefits:

- WSDL document centralizes update of the web services development workflow by allowing, in case of module interface changes, these changes to be modified in one document.
- Implementation codes of the web service endpoint which correspond to the updates can be generated dynamically on the fly.
- It provides a better overview of the whole interface and the methods which are exported as web service methods.

For these reasons, the development of the server module for CCMSs has adopted the top-down development approach. The WSDL document of the web services is generated by a WSDL generator which can be plugged into and run in the CCMS compiler backend.

In a nutshell, the mapping of module methods to WSDL element artifacts takes these guidelines which are summarized in table 3.3.

Module interface elements	**translate to WSDL elements**
generic asset types	XML Schema *simpleType* or *complexType* which are utilized by other WSDL elements as type references
concrete asset types	XML Schema *complexType*
module method name	a pair of request and response WSDL *message* elements with the method name as the naming prefix
module method signature	name of method is preserved and maps to *name* attribute of a WSDL *operation* element
module method parameters	generally map to WSDL *part* elements within a WSDL request *message* element
module method return type name	generally maps to WSDL *part* element with a WSDL response *message* element
module method concrete signature	maps to *name* attribute of the WSDL *operation* elements within the *binding* element.

Table 3.3: Mapping of generic module elements to WSDL elements

3.2.2 Functional Requirements of a Server Module

The functional requirements of a server module are:

1. The WSDL document that describes the web services of the server module must be generated using a generator.
2. The server module must be able to listen on a network port and accept requests.
3. The server module must be able to handle web services request messages and produce response messages that are SOAP conformed.
4. Serialization of the exchanged messages in XML should be a built-in feature of the server module.
5. The server module must comply with the *module interface* definition of a CCMS module.
6. The request messages of the web services tier must be mapped to method calls of the *module interface*. The method calls are delegated to pre-configured base modules of CCMSs.

3.2.3 Functional Requirements of Web Technologies

Since web services endpoint is deployed in connection with many of the fundamental web technologies nowadays, there are several issues concerning the choice of these technologies that are relevant to the provision of web services.

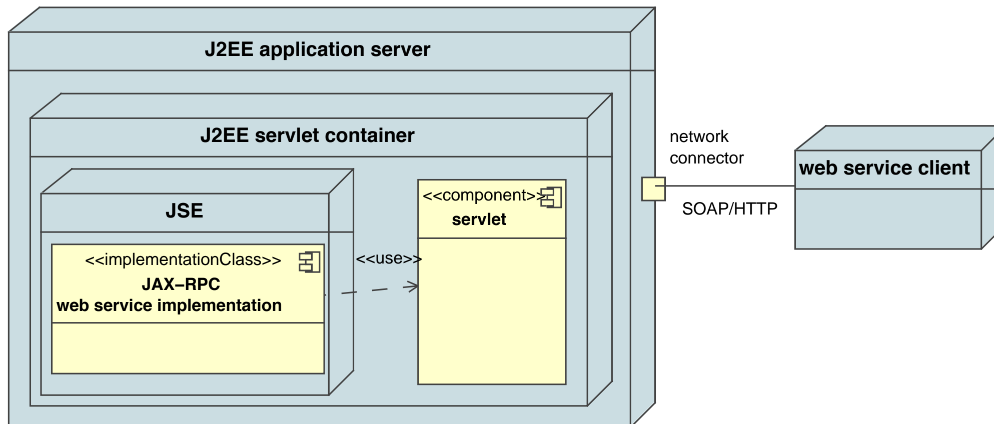


Figure 3.8: JSE web service deployment using servlet delegation

The Java enterprise platform provides corresponding technologies that support web services implementation and deployment [Mah04]. There are mainly two types of web services endpoints [SBMS04] on this platform:

- base on binding the Java servlet technologies, JSE[10] deployment,
- base on binding a EJB[11] as the endpoint implementation.

Both type of these endpoints have found widespread use in the arena of Java web services implementation.

In the first type of endpoint, a definition of the *endpoint interface* and an *implementation class* are provided. The *endpoint interface* contains definitions of web service operations in form of Java methods. The *implementation class* contains code which implements these endpoint methods. In terms of the runtime environment to support a JSE, JSEs are generally deployed into a web servlet container, and have access to the same resources and context information that a servlet has. When a JSE is deployed, a JAX-RPC servlet provides the hosting environment for the *implementation class* and is responsible for responding to HTTP-based SOAP requests, parsing the SOAP messages, and invoking the corresponding methods of the JSE *implementation class*. When the JSE returns a value from the method invocation, the JAX-RPC servlet creates a SOAP message to hold the return value or a SOAP fault if an exception occurs in any operation. It then sends that SOAP message back to the requesting client via an HTTP reply message. The structural setup is briefly illustrated in figure 3.8.

Because a JSE is embedded in a servlet which itself resides inside a servlet container or other servlet-standard conformed web container, it can access the same resource that the servlet can. The difference turns out that either the resource resides locally or remotely, the latter cannot be accessed effectively unless a remote object reference is present or can be obtained. For the purpose of obtaining remote object references, a proxying, naming and lookup mechanism, such as JNDI[12] for naming can be used. Servlet context and session information can theoretically be accessed also. In addition, one advantage of this simple binding model inside the servlet allows access to actual SOAP message including the method parameters or the header part of the message. JSE web service endpoints are straightforward to implement based

[10] JSE – JAX-RPC Service Endpoint
[11] EJB – Enterprise JavaBean
[12] JNDI – Java Naming and Directory Interface

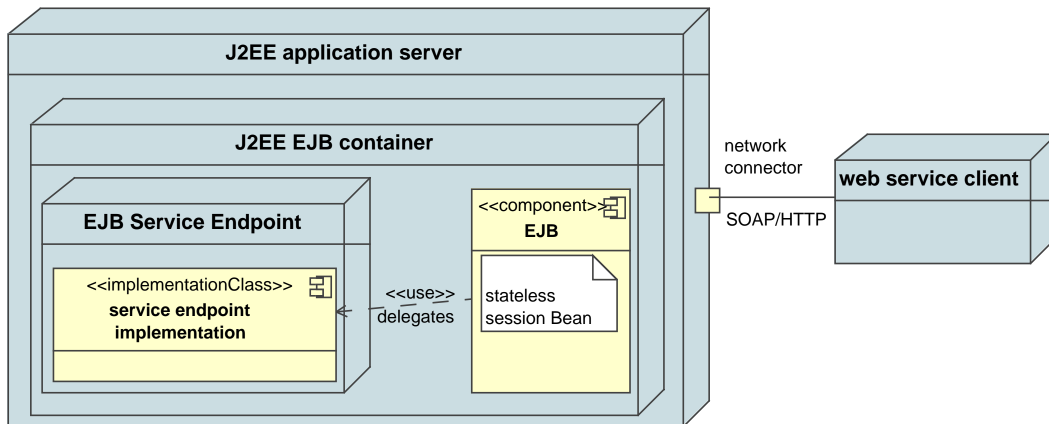


Figure 3.9: Web services endpoint deployed with a stateless EJB component

on the servlet model. Moreover, the servlet programming model is widespread which makes the deployment of the JSE web services endpoint in standard servlet container efficient.

In the EJB type of service endpoint, a stateless *session bean* is made accessible as a web service by deploying the bean with a *remote* and a *local interface*. An *endpoint interface* which extends the `java.rmi.Remote` interface is defined. An EJB endpoint can be a specific new stateless session bean developed to serve a web service endpoint, or an exisiting stateless session bean can be re-deployed as an endpoint. The JAX-RPC specification [jcp06, Mic06] explains the steps to turn such a stateless session into a web service endpoint.

A SOAP client uses a WSDL document associated with the EJB endpoint to send a SOAP message to the application server that hosts the EJB endpoint. The following figure 3.9 depicts a stateless EJB endpoint processing SOAP messages.

In terms of the EJB runtime environment, an EJB can access resources, other EJBs, and web services using JNDI. In addition, an EJB can interact with its container via callback methods and the bean SessionContext interface. Generally, transaction and security is managed by the container.

With regard to the possible implementation of the service endpoint mentioned, the web-tier poses these requirements on the web technologies used to transport and host the web service:

1. A stateless protocol such as HTTP should be used.

2. Because of the simplicity and statelessness of servlets, it suggests that a JSE should be implemented with servlet technology.

3. Since the server module must be able to serve client requests and produce responses itself, it must process the functionality of an HTTP server.

4. The implementation of the endpoint should process SOAP messages seamlessly.

5. For coherent reason with CCMSs, the JSE should adhere to the JAX-RPC specifications when it comes to the details of the endpoint implementations.

3.2.4 Non-functional Requirements

Regarding to the nature and characteristics of the Java programming language, in which the modules and components of concept-oriented content management sys-

tems are implemented. The web services layer is consequently implemented in the same language or has adopted web services frameworks which are based on the Java programming language. Some of the following non-functional requirements are conceivable:

1. Depending on the configuration of the runtime CCMS module configuration, the processing time of request and the creation of response should meet certain time-limits requirements to guarantee an acceptable response rate of the system.
2. Threading support of handling multiple client requests without blocking the server module.
3. The performance of the integrated web services frameworks should be fine-tuned to scale to increasing request load.

3.3 Web Services Technologies Survey

In the development of web services, there are mainly two concurring platforms supporting the entire need and development workflows of web services. They are the .NET and the J2EE web services.

The .NET platform is designed for close compatibility with the Windows operating system, and it takes full advantage of native Windows features. On the other hand, the J2EE platform takes advantage of the Java virtual machine's portability layer to provide the same features and functionality across all operating systems on which it runs.

Because of key differences, interoperability between the .NET platform and the J2EE platform is limited and since the J2EE platform web services are selected for the server module implementation. The evaluation of web services relevant software technologies will solely concentrate on the J2EE platform.

3.3.1 Java based Web Services Technologies

The Java APIs for XML fall into two broad categories: those that deal explicitly with the processing of XML documents and those that deal with the procedures used to interchange XML-based documents. There are three document-oriented Java XML APIs:

- Java API for XML Processing (JAXP)
- Java Architecture for XML Binding (JAXB)
- SOAP with Attachments API for Java (SAAJ)

There are two APIs that deal with the procedures used to interchange XML-based documents for Java web services:

- Java API for XML-based RPC (JAX-RPC)
- Java API for XML Registries (JAXR)

JAXP is invaluable for processing XML documents. It is a powerful and extensible API based on the concept of using external plug-in modules. JAXP supports SAX and DOM with full XML namespace processing and parsing. All XML parsers verify that an XML document is well formed and syntactic correctness. Some XML parsers are known as validating parsers. Validating parsers, in addition to verifying the structure of an XML document, also validate the contents of a well-formed XML

document against the appropriate DTD or XML schema. Xerces of the Apache Software Foundation, is a widely used example of an XML validating parser that supports both the SAX and DOM APIs.

JAXB is another piece of XML schema oriented processing software by which Java applications can process XML documents which conform to an XML schema document. It is a Java technology that enables developers to easily generate Java classes by creating objects in the form of JavaBeans from an XML schema. Developers can use JAXB to create a representation of an XML schema in terms of Java code. It provides an easy and convenient way to incorporate XML data and XML-related processing functionality into Java applications without deep knowledge about the intricacies and mechanics of XML.

SAAJ is a different API than JAXP or JAXB. Rather than being an API related to processing XML documents, SAAJ is a SOAP-related API. It provides a standard means by which Java applications can send XML documents by using SOAP. SAAJ is obviously targeted at web services related applications in which a low-level manipulataion of the SOAP message within the program logic is desired.

The above mentioned APIs are all more or less XML document oriented, in terms of web services development, not only a more sophisticated software framework for handling XML documents is important; but also the functionalities of message creation, delivery, serialization and deserialization are critical. In this respect, JAX-RPC is a web services oriented API emphasizing on the procedural side of web services development rather than solely on the document-related side. JAX-RPC [Mic06] is an RPC invocation mechanism in a nutshell. It enables Java applications to invoke XML-based RPC operations conforming with the SOAP 1.2 specification. It enables Java software to participate within a web services application. In terms of programming, either as web services provider, i.e. server side web services endpoints, or as applications, i.e. client side web services client applications, invoking Java or non-Java-based[13] web services.

JAXR is a uniform and standardized Java API for accessing and querying different kinds of XML registries with UDDI and the OASIS proposed ebXML registries. It provides Java applications with a unified information model that describes the content and meta-data included within XML registries. Java developers can develop registry client software with JAXR to query, publish or update UDDI registries. The JAXR API is now an integral parts of the J2EE version 1.4 platform.

There is a useful bundle of Java web services development software packaged by Sun Microsystem under the name *Java Web Services Developer Pack (Java WSDP)*. Java WSDP is a free integrated toolkit to help Java developers create any type of XML-related software with the latest Java APIs. It can be downloaded from java.sun.com. Web services related software developed with Java WSDP is assured to conform to WS-I[14] basic profile 1.0. In addition to offering JAXP, JAXB, SAAJ, JAX-RPC, and JAXR, The Java WSDP development package also includes features, such as JavaServer Faces (JSF) and access to all of the J2EE security features (including authentication and encryption).

[13]examples are document literal web services which are not RPC operations in conventional sense
[14]WS-I – web services interoperability organization

Chapter 4

WSDL Generator and Web Services Modules Design

This chapter describes the design of the WSDL generator and the web services server module. A description of the mapping strategies for modeling the *module interface* in WSDL is given in section 4.1. The WSDL generator design issues are discussed in section 4.2. The design issues of the server module as a web services endpoint are explained with several possible architectural design approaches in section 4.3. An emphasis is given to the juxtaposition of the different design approaches to each other with discussions on the pros and cons regarding the specific design, in order to find an advantageous disign approach.

4.1 Web Services Interface Description — WSDL

One of the main functional requirements of the server module is the provision of the asset CRUD operations as web services operations. The *module interface* defined in the Java interface `Module` of the `de.tuhh.sts.cocoma.generic` package contains the methods for the CRUD operations to be exported as web services operations. The definition of this Java interface is given in code example 4.1.1.

The first group of methods of the interface definiton consists of the module lifecycle methods; they are responsible for the *initialization* (implemented by the `init` method), *activation* (implemented by the `start` method) and *deactivation* (implemented by the `stop` method) of the server module. The other groups of interface methods are characterized by the manipulative operations on asset instances. Each group of these methods consists of at least one instance of an *overloaded* method to the corresponding method declaration. The difference among the overloaded methods is obvious with regard to the method signatures. The purpose of having overloaded methods consists in the fact that different method parameters, which are assets, are allowed as method arguments for the invocation of these overloaded methods.

Regarding to the functional requirements of the web services endpoint, the task of the server module is to provide web service enabled access to these interface methods for the manipulation of asset instances of a CCM subsystem. In a nutshell, there are four groups of these interface methods which must be translated into web services operations. These are methods for the creation, retrieval, modification and destruction of asset instances as mentioned in section 3.2 of chapter 3. According to the JAX-RPC web services specification, a Java interface containing the exported methods must extend the `java.rmi.Remote` interface and these methods must throw an `java.rmi.RemoteException` to denote an exceptional event in case of a failure. These interface methods are listed in table 4.1.

```
public abstract interface de.tuhh.sts.cocoma.generic.Module {

    // module lifecycle methods
    public void init(Component component,
                     java.lang.String name,
                     java.util.Map parameters)
    public void start()
    public void stop()
    public void addInstancesLifeCycleListener(LifeCycleListener listener)
    public void removeInstancesLifeCycleListener(LifeCycleListener listener)
    public AssetClass getClass(java.lang.String name)


    // module asset creation mothods
    public Asset create(AssetClass assetClass,
                        AbstractAsset prototype)
    public AssetIterator create(AssetClass c,
                                AssetIterator ps)
    public Asset create(AssetClass assetClass,
                        Module.MemberInitialization[] initialization)


    // module asset deletion methods
    public NewAsset delete(Asset asset)
    public AssetIterator delete(AssetIterator assets)


    // module asset query methods
    public AssetIterator lookfor(AssetClass assetClass,
                                 AbstractAsset prototype)
    public AssetIterator lookfor(AssetClass assetClass,
                                 AssetIterator prototypes)
    public AssetIterator lookfor(AssetClass assetClass,
                                 Module.QueryConstraint[] constraints)
    public AssetIterator lookfor(AssetClass assetClass,
                                 java.lang.String retrievalExpression)
    public Asset lookfor(ID assetID)
    public AssetIterator lookfor(java.lang.String retrievalExpression)


    // module asset modification methods
    public Asset modify(Asset asset,
                        AbstractAsset prototype)
    public AssetIterator modify(AssetIterator assets,
                                AbstractAsset prototype)
    public AssetIterator modify(AssetIterator assets,
                                Module.MemberInitialization[] initialization)
    public Asset modify(Asset asset,
                        Module.MemberInitialization[] initialization)

}
```

Code 4.1.1: Definition of the **Module** Java interface

Method group	Method sigatures
Asset creation	*public* Asset **create**(AssetClass acls, AbstractAsset abstasst) **throws** RemoteException
	public AssetIterator **create**(AssetClass acls, AssetIterator asstiter) **throws** RemoteException
	public Asset **create**(AssetClass acls, MemberInitialization[] mmbinit) **throws** RemoteException
Asset deletion	*public* NewAsset **delete**(Asset asst) **throws** RemoteException
	public AssetIterator **delete**(AssetIterator asstiter) **throws** RemoteException
Asset modification	*public* Asset **modify**((Asset asst, AbstractAsset abstasst) **throws** RemoteException
	public AssetIterator **modify**(AssetIterator asstiter, AbstractAsset abstasst) **throws** RemoteException
	public AssetIterator **modify**(AssetIterator asstiter, MemberInitialization[] mmbinit) **throws** RemoteException
	public Asset **modify**(Asset asst, MemberInitialization[] mmbinit) **throws** RemoteException
Asset query	AssetIterator **lookfor**(AssetClass acls, AbstractAsset abstasst) **throws** RemoteException
	public AssetIterator **lookfor**(AssetClass acls, AssetIterator asstiter) **throws** RemoteException
	public AssetIterator **lookfor**(AssetClass acls, QueryConstraint[] qryconst) **throws** RemoteException
	public AssetIterator **lookfor**(AssetClass acls, java.lang.String str) **throws** RemoteException
	public Asset **lookfor**(ID id) **throws** RemoteException
	public AssetIterator **lookfor**(java.lang.String str) **throws** RemoteException

Table 4.1: Asset methods exposed in the web service interface definition

4.1.1 Mapping the Module Interface to WSDL

In section 3.2.1 of chapter 3, an overview for translating the module interface methods to WSDL elements and artifacts is summarized in table 3.3. These generalized mapping rules are now applied to the design mapping process.

In order to map the module interface methods correctly, the order of the translation of the method elements to the corresponding constituents of a WSDL document is important and should be observed:

1. map the necessary generic types of the `de.tuhh.sts.cocoma.generic` package to the corresponding XML Schema types[1],

2. map the concrete asset type structure of the asset classes in a asset domain model definition to the corresponding XML Schema types,

3. extract the parameter list of each method and map each argument of the method to a **wsdl:part** element within a **wsdl:message** element; inside the **wsdl:part** element, the argument name is mapped to the **name** attribute and the argument type is mapped to the corresponding XML schema type using the **type** attribute, a snippet of the mapped parts is shown in code 4.1.2,

4. extract the parameter list of each method, arguments are translated to **wsdl:part** elements; arguments are grouped inside the corresponding *request* **wsdl:message** element, an excerpt of code is shown in code example 4.1.3,

5. extract the return type of each interface method and map the return type to a **wsdl:part** element within a *response* **wsdl:message** element; map the return type name to the **name** attribute and the return type to the corresponding XML schema type, code excerpt is shown in code example 4.1.4,

6. in each method group, the overloaded methods are treated differently in the mapping process. Since there is no way to identify a method without the knowledge of the method signature. The mapping of the method name alone does not suffice the information need. Therefore overloaded methods are mapped in the **wsdl:operation** elements of the **wsdl:portType** element by differentiating between the **wsdl:input** and **wsdl:output** elements. A code excerpt is shown in code example 4.1.5,

7. concrete binding of each method is mapped to the corresponding **wsdl:operation** element inside the **wsdl:binding** element. The request and response messages that belong to the mapped method are bound to a concrete protocol binding and given in the **wsdl:input** and **wsdl:output** elements of the WSDL document. A code excerpt is shown in code example 4.1.6.

```
<wsdl:message name="lookforRequest">
    <wsdl:part name="argument0" type="impl:AssetClass"/>
    <wsdl:part name="argument1" type="impl:AbstractAsset"/>
</wsdl:message>
```

Code 4.1.2: Mapping of interface method arguments

Overloaded methods can be identified using the above mentioned mapping strategy without the need to alter the names of the overloaded interface methods themselves. This approach is not the only mapping possibility, the other way to map overloaded interface methods is to provide unique naming to each overloaded interface method.

[1] details on XML Schema mapping see section 4.1.2

```
<wsdl:message name="lookforRequest_1">
    <wsdl:part name="argument0" type="impl:ID"/>
</wsdl:message>
```

Code 4.1.3: Mapping of argument within request message

The actual design should prefer the first approach than the latter one because there is the need to preserve the module interface definition. If the unique naming approach is adopted, the naming of the overloaded methods cannot be used uniquely without naming each method differently; however, this approach is contradictory to the initial requirements for the interface translation.

4.1.2 Modeling Generic Asset Types with XML Schema

The initial mapping process begins with the mapping of generic types of the interface hierarchy to XML Schema. The complete schema document is shown in appendix A on page 89.

The process of modeling the asset types is essentially a twofold task:

- generic modeling - construct XML schema types to model the interface hierarchy contained in the generic package,
- asset modeling - translate the class definitions of concrete asset models into the corresponding XML schema language types.

Generic Modeling

The first modeling task begins with the conversion of the interface hierarchy types of the `de.tuhh.sts.cocoma.generic` package. The modeling strategy of the the generic types is to translate each generic type into a corresponding `complexType` of the XML schema language. While simple generic types can be translated directly to straightforward XML schema types, other composite generic types which are made up of other generic types must be treated differently. The important issue is that these composite types contain references to one or more of other simple or composite generic types. In order to model the composite types correctly, their references to other generic types are mapped to multiple XML schema `elements` with the corresponding generic type defined using the *type* attribute of the specific `element`.

During the design phase, it has been discovered that the generic types themselves are rather invariant in terms of their resemblance to their corresponding XML schema language types. This fact is important to notice in the design process. For this reason, all the interface types in the generic interface hierarchy have been modeled in the XML schema language once. The resulting XML schema document is extracted and modularized for reuse. The fact that this XML schema modeling is referenced in the WSDL document which describes the module interface in the server module proves that the modeling fulfills the requirements of schema document reuse.

Another point worth mentioning is the decision to model all the generic *interfaces* together with their sub-interfaces in the generic package as *concrete* XML schema types during the design phase. The subtle difference between interface and class in the Java language has produced discrepancy and semantics mismatch if this programming language concepts are mapped directly into the XML schema language. The reason is that interface types such as *Asset*, *AssetClass*, etc. represents abstract type contracts which cannot be instantiated. While only classes representing concrete types can be instantiated, once these concrete types has implemented the interface types. Speaking of modeling concept, concrete class types represent subtypes of their implemented interfaces respectively. Although it is possible to define *abstract complexTypes* as a

```
<wsdl:message name="lookforResponse_1">
    <wsdl:part name="lookforReturn" type="impl:Asset"/>
</wsdl:message>
```

Code 4.1.4: Mapping of method return type

modeling concept using the XML schema language [Con04b], in terms of modeling, the concept and semantics of this specific XML schema language construct is nevertheless different from the programming language concept of an abstract type because the *abstract simpleType* or *abstract complexType* cannot be instantiated in an XML instance document. Such *abstract* definitions of XML schema types serve the sole purpose of ensuring the modeling completeness of the language. Since the generic interface types must be used in the parameter list and the return types of the *module interface* methods, mapping them directly with their programming language interface semantics into their *abstract* counterpart in the domain of XML schema modeling will render the types in the web services interface definition not instantiatable; therefore it is not an appropriate modeling approch. The design solution for this semantics mismatch turns out that it can open up the type system confinement by bridging the two antipodes – to model interface types as *complexTypes* with the XML schema language. Taking this measure further, concrete asset types can be used to substitute for the generic interface types with a concept called *substitutionGroup* [Con04b] of the XML schema language.

While the strategy concerning type conversion is mentioned above, the design must also deal with public methods of the interface types. In terms of the adequacy of a model, the methods contained in an interface type, such as those methods of the generic interfaces, must be extracted. The reason for this is that every object, for instance, one that represents a parameter or a return type of a web service method implementation, must be instantiated retrospectively by the object-oriented language runtime environment. According to the JAX-RPC specification [Mic06], this means that the complete set of *accessor methods*[2] of a class that implements a specific generic interface must be modeled as accessor methods of a full-fledged *JavaBean object* for the web services. Therefore those properties of the generic interface definitions which represent the properties of a *JavaBean* implementation class must be extracted by selecting the *accessor methods* declarations in the generic interface definitions. The *JavaBean* properties must be modeled as XML schema *simpleType* or *complexType* within the XML schema model.

Asset Modeling

The second type of modeling is concerned with the conversion from asset model definitions to concrete XML schema types. It is a fundamental task in the mapping process because the constituents of the asset model – the concrete asset classes must be translated from the asset model to an XML schema information model without loosing information of the domain.

In code example 4.1.7, an asset model called *EquestrianModel* is defined which contains only one asset class *Equestrian*.

This asset class contains a content and concept pair as described in section 2.1.2 of chapter 2. In a nutshell, an asset class can be translated into a *complexType* in the XML schema information model, for instance, the class *Equestrian* becomes the *complexType* element containing the *name* attribute with the value *Equestrian*. The subelements are mapped directly from the *characteristic* fields of the *concept* part of the class. If the type of the fields is of an elementary nature and has a

[2] accessor methods are also known as getter and setter methods

```
<wsdl:portType name="CocomaGenericSModulePortType">
    <wsdl:operation name="getClass" parameterOrder="argument0">
        <wsdl:input name="getClassRequest" message="impl:getClassRequest"/>
        <wsdl:output name="getClassResponse" message="impl:getClassResponse"/>
    </wsdl:operation>
    <wsdl:operation name="create" parameterOrder="argument0 argument1">
        <wsdl:input name="createRequest_1" message="impl:createRequest_1"/>
        <wsdl:output name="createResponse_1" message="impl:createResponse_1"/>
    </wsdl:operation>
    <wsdl:operation name="create" parameterOrder="argument0 argument1">
        <wsdl:input name="createRequest_2" message="impl:createRequest_2"/>
        <wsdl:output name="createResponse_2" message="impl:createResponse_2"/>
    </wsdl:operation>
    <wsdl:operation name="create" parameterOrder="argument0 argument1">
        <wsdl:input name="createRequest_3" message="impl:createRequest_3"/>
        <wsdl:output name="createResponse_3" message="impl:createResponse_3"/>
    </wsdl:operation>
</wsdl:portType>
```

Code 4.1.5: Mapping of overloaded methods

```
<wsdl:binding name="binding" type="tns:serverporttype">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="rpc"/>
  <wsdl:operation name="create">
    <soap:operation soapAction="default" style="rpc"/>
    <wsdl:input name="createRequest_1">
      <soap:body use="encoded"/>
    </wsdl:input>
    <wsdl:output name="createResponse_1">
      <soap:body use="encoded"/>
    </wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="create">
    <soap:operation soapAction="default" style="rpc"/>
    <wsdl:input name="createRequest_2">
      <soap:body use="encoded"/>
    </wsdl:input>
    <wsdl:output name="createResponse_2">
      <soap:body use="encoded"/>
    </wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="create">
    <soap:operation soapAction="default" style="rpc"/>
    <wsdl:input name="createRequest_3">
      <soap:body use="encoded"/>
    </wsdl:input>
    <wsdl:output name="createResponse_3">
      <soap:body use="encoded"/>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
```

Code 4.1.6: Mapping of interface method in the binding element

```
model EquestrianModel

class Equestrian {
    content reproduction                  : String
    concept
      characteristic yearOfCreation       : java.util.Calendar
      characteristic medium               : String
      characteristic location             : String
      characteristic background           : String
      characteristic portrayal            : String
      characteristic numberOfSubject      : java.lang.Integer
      characteristic numberOfDimensions   : java.lang.Integer
      relationship painter                : Artist
      relationship epoch                  : Epoch
}
```

Code 4.1.7: Asset model definition of class Equestrian

counterpart in the XML schema language information model, that specific type is mapped directly to the corresponding XML schema type, for instance, the asset *characteristic* `yearOfCreation` has the *java.util.Calendar* type; it can be mapped to the *xsd:datetime* schema type. Another *characteristic* `portrayal` has the type *String* which can be mapped to the *xsd:string* schema type. Moreover, the type *java.lang.Integer* of the *characteristic* `numberOfSubject` and `numberOfDimensions` can be mapped to the *xsd:integer* schema type. A special case of mapping is characterized in the mapping of *relationship* within an asset class definition. Generally, a *relationship* is mapped to an element in the XML schema document which has the *type* attribute *xsd:ID*. The reason for this strategy is explained by the fact that asset instances can be queried using their asset identifiers – IDs. A *relationship* referencing another asset instance within an asset class definition is simply a reference to another asset intance which is addressable using the ID of this asset instance. The complete converted XML schema document which reflects the `Equestrian` asset class definition is given in the code example 4.1.8.

In a further survey of the mapping possibilities, an asset class definition can be extended in the sense that one asset class extends from another asset class using the *refines* keyword in an asset inheritance definition. The snippet of code in 4.1.9 shows a model definition called `gkns` illustrating an inheritance relationship between these asset classes.

The model `gkns` shown in code 4.1.9 has three asset classes: the `Fund` class is a *base class* or *super class*; while the other asset classes: `Dokument` and `Korrespondenz` are inherited from the `Fund` class using the *refines* keyword. This asset model is converted to the corresponding XML schema information model which is shown in code 4.1.10 by using the general rules described previously. The difference to the previous asset model example is characterized by the translation of the inheritance relationship within an asset model into the *type derivation* mechanism of the XML schema information model [Con04a]. The mechanism used to denote a type derivation is the *extesion* keyword which is followed by the *base* attribute containing the name of the base type to be derived from.

Substituting Elements

The XML schema language provides a mechanism, called *substitution groups*, using the *substitutionGroup* keyword to allow some elements to be substituted by another elements within an XML schema instance document. More specifically, elements can be assigned to a special group of elements that are able to substitute for a particular

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema targetNamespace="urn:de.tuhh.sts.cocoma.generic"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:cocoma="urn:de.tuhh.sts.cocoma.generic"
           xmlns:xdt="http://www.w3.org/2003/05/xpath-datatypes"
           xmlns:tns="urn:de.tuhh.sts.cocoma.generic">
    <xs:element name="ArtistList" type="tns:ArtistList"/>
    <xs:element name="EpochList" type="tns:EpochList"/>
    <xs:element name="Equestrian" type="tns:Equestrian"/>
    <xs:element name="Artist" type="tns:Artist"/>
    <xs:element name="Epoch" type="tns:Epoch"/>
    <xs:element name="EquestrianList" type="tns:EquestrianList"/>
    <xs:complexType name="ArtistList">
     <xs:sequence>
      <xs:element ref="tns:Artist" minOccurs="0" maxOccurs="unbounded"/>
     </xs:sequence>
    </xs:complexType>
    <xs:complexType name="EpochList">
     <xs:sequence>
      <xs:element ref="tns:Epoch" minOccurs="0" maxOccurs="unbounded"/>
     </xs:sequence>
    </xs:complexType>
    <xs:complexType name="Equestrian">
        <xs:complexContent>
         <xs:extension base="cocoma:Asset">
          <xs:sequence>
            <xs:element name="numberOfDimensions" type="xs:integer"/>
            <xs:element name="location" type="xs:string"/>
            <xs:element name="yearOfCreation" type="xs:dateTime"/>
            <!-- other characteristic-elements are masked out for brevity -->
            <xs:element name="painterRef" type="xs:ID" minOccurs="0"/>
            <xs:element name="epochRef" type="xs:ID" minOccurs="0"/>
          </xs:sequence>
         </xs:extension>
        </xs:complexContent>
    </xs:complexType>
    <xs:complexType name="Artist">
       <xs:complexContent>
         <xs:extension base="cocoma:Asset">
           <xs:sequence></xs:sequence>
         </xs:extension>
       </xs:complexContent>
    </xs:complexType>
    <xs:complexType name="Epoch">
       <xs:complexContent>
         <xs:extension base="cocoma:Asset">
           <xs:sequence></xs:sequence>
         </xs:extension>
       </xs:complexContent>
    </xs:complexType>
    <xs:complexType name="EquestrianList">
     <xs:sequence>
      <xs:element ref="tns:Equestrian" minOccurs="0" maxOccurs="unbounded"/>
     </xs:sequence>
    </xs:complexType>
</xs:schema>
```

Code 4.1.8: XML schema representing the Equestrian class

```
model gkns

class Fund {
    content
        contentIds                      : String
    concept
        characteristic titel            : String
        characteristic datum            : java.util.Calendar
        characteristic bemerkung        : String
        characteristic erfassungsdatum : java.util.Calendar
        characteristic aenderungsdatum : java.util.Calendar
        characteristic typ              : String
        relationship standort           : Referenz
        relationship erfasser           : User
        relationship verschlagwortung  : Schlagwort*
        relationship kommentare         : Kommentar*
        relationship masks              : Mask*
}

class Dokument refines Fund {
    concept
        characteristic inhalt                 : String
        characteristic entstehungsort         : String
        characteristic umfang                 : String
        characteristic sperrvermerkJuristisch : java.util.Calendar
        characteristic sperrvermerkFachlich   : java.util.Calendar
}

class Korrespondenz refines Dokument {
    concept
        characteristic absender              : String
        characteristic absenderInstitution : String
        characteristic adressat              : String
        characteristic adressatInstitution : String
        characteristic betreff               : String
}
```

Code 4.1.9: Multiple asset class definitions with inheritance

element which is called the *head element*. Substituting elements in a *substitution group* must have the same type as the type of the *head element* or they must have a type that is derived from the type of the *head element*. In the mapped XML schema document, according to these mentioned premises about the substituting element type in a *substitution group*, the mapping design must outline the following type and type derivation rules if the mapping involves the conversion of a class inheritance relationship of an asset model to the corresponding XML schema model, an excerpt of the schema document is given in code 4.1.11.

As it is shown in the schema document excerpt, the `Fund` class is mapped to a schema element called `Fund` with type `Fund` which is defined as a *complexType* in the schema document. The `Fund` element can substitute the generic *head* element – `Asset` which is the *head* of the *substitution group*. The masked out part of the definitions of the *complexTypes* in the example is basically identical to the *complexType* definitions in the previous schema document as shown in code 4.1.10.

It is worth to point out that the type `Fund` must be a derived type of the type `Asset` in order to allow the *substitution group* mechanism to function properly. This premise also applies true to the type `Dokument`, which is a derived type from `Fund`;

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema targetNamespace="urn:de.tuhh.sts.cocoma.generic"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:tns="urn:de.tuhh.sts.cocoma.generic"
           xmlns:cocoma="urn:de.tuhh.sts.cocoma.generic">
    <xs:element name="Fund" type="tns:Fund" substitutionGroup="cocoma:Asset"/>
    <xs:element name="FundList" type="tns:FundList"/>
    <xs:element name="Dokument" type="tns:Dokument"/>
    <xs:element name="Korrespondenz" type="tns:Korrespondenz"/>
    <xs:complexType name="Fund">
     <xs:complexContent>
      <xs:extension base="cocoma:Asset">
       <xs:sequence>
        <xs:element name="typ" type="xs:string"/>
        <xs:element name="datum" type="xs:dateTime"/>
        <xs:element name="titel" type="xs:string"/>
        <!-- other characteristic-elements are masked out for brevity -->
        <xs:element name="erfasserRef" type="xs:ID" minOccurs="0"/>
        <xs:element name="verschlagwortungRef" type="xs:ID" minOccurs="0"/>
       </xs:sequence>
      </xs:extension>
     </xs:complexContent>
    </xs:complexType>
    <xs:complexType name="FundList">
     <xs:sequence>
      <xs:element ref="tns:Fund" minOccurs="0" maxOccurs="unbounded"/>
     </xs:sequence>
    </xs:complexType>
    <xs:complexType name="Dokument">
     <xs:complexContent>
      <xs:extension base="tns:Fund">
       <xs:sequence>
        <xs:element name="entstehungsort" type="xs:string"/>
        <xs:element name="sperrvermerkFachlich" type="xs:dateTime"/>
        <xs:element name="sperrvermerkJuristisch" type="xs:dateTime"/>
        <xs:element name="inhalt" type="xs:string"/>
        <xs:element name="umfang" type="xs:string"/>
       </xs:sequence>
      </xs:extension>
     </xs:complexContent>
    </xs:complexType>
    <xs:complexType name="Korrespondenz">
     <xs:complexContent>
      <xs:extension base="tns:Dokument">
       <xs:sequence>
        <xs:element name="adressatInstitution" type="xs:string"/>
        <xs:element name="absenderInstitution" type="xs:string"/>
        <xs:element name="betreff" type="xs:string"/>
        <xs:element name="adressat" type="xs:string"/>
        <xs:element name="absender" type="xs:string"/>
       </xs:sequence>
      </xs:extension>
     </xs:complexContent>
    </xs:complexType>
</xs:schema>
```

Code 4.1.10: XML schema modeling the inheritance relationship

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema targetNamespace="urn:de.tuhh.sts.cocoma.generic"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:tns="urn:de.tuhh.sts.cocoma.generic"
           xmlns:cocoma="urn:de.tuhh.sts.cocoma.generic">
    <xs:element name="Fund" type="tns:Fund" substitutionGroup="cocoma:Asset"/>
    <xs:element name="FundList" type="tns:FundList"/>
    <xs:element name="Dokument" type="tns:Dokument"
                substitutionGroup="tns:Fund"/>
    <xs:element name="Korrespondenz" type="tns:Korrespondenz"
                substitutionGroup="tns:Dokument"/>
    <complexType name="Fund">
      <!--
        ... complexType definitions are masked out ...
       -->
    </complexType>
</xs:schema>
```

Code 4.1.11: XML schema excerpt with substitution group

and type `Korrespondenz` which is a derived type from type `Dokument`. This simple inheritance hierarchy is shown in figure 4.1.

The main reason to use the *substitution group* mechanism is to preserve the homogeneity of the module interface while ensuring a certain degree of flexibility in the design of the web services interaction layer. Since interface methods takes generic parameters; together with the return types that are also generic interface types, an object-oriented language such as Java can take any subtypes of the interface types as long as they represent classes implementing the generic arguments or return types. However, the type derivation mechanism is not exactly the same in a modeling language such as the XML schema language. This discrepancy in the language semantics has been mentionen previously. Thusly in terms of type coherence, a design that is adequate for the web services interface methods must take these mapping issues into account. This is where the *substitution group* mechanism comes in – by making concrete asset types as subtypes of generic base types such as the example shown previously, the total number of methods remains the same without having to provide a new method for every known concrete asset subtype method arguments because otherwise a new method must be provided for every concrete subtype.

Subtypes can be passed as arguments to the methods as long as they can *substitute* the *head* element, as the example in figure 4.1 shows, the *head* element for `Korrespondenz` is `Dokument`; the *head* element for `Dokument` is `Fund`. The base type `Fund` can substitute the generic base type `Asset`. This design approach bridges the discrepancy gap without violating the semantical rules of asset language modeling.

4.1.3 Generation of XML Schema Definitions

The responsibility of producing an XML schema document for the web services endpoint remains a major issue in the design. The task takes a generative approach by using an XML schema generator of the compiler backend [Bos04, Bos03, BSHS06].

The XML schema generator is compliant to the backend generator interface specification. It can be configured to generate XML schema document from a model of asset definition language as shown in code example 4.1.7 and 4.1.9. The XML schema generator is capable of generating both flavor of XML schemas: schema documents with or without using the *substitution group* mechanism. Configuration examples of running the XML schema generator in both modes are given in appendix B on page 94 and 95.

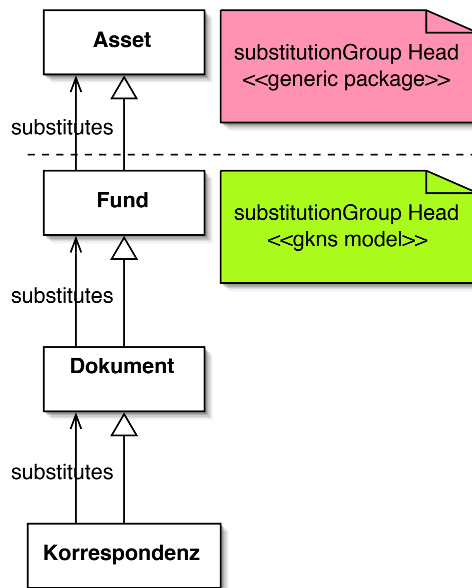


Figure 4.1: Asset type inheritance

4.2 WSDL Generator

For the purpose of web services development and deployment, a WSDL document must be produced. By adopting a generative approach, generating the WSDL document that describes the module interface has led to the development of a WSDL generator software component that can be integrated into the CCMS compiler backend in order to produce the required WSDL document.

4.2.1 Design Overview

The WSDL generator must be designed according to the generator API specification. It is run inside the compiler backend. By taking advantage of the *generator API symbol table* and the *intermediate model*, different generators can communicate with each other. This communication pattern over the passing of information using the *intermediate model* is common in compiler design.

The *API symbol table* has no concern to the WSDL generator since it will not use the information within the table to generate the constituent of the WSDL document. However, it is worth mentioning that the *intermediate model* which is created by the compiler parser represents an internal object model of the asset model. This information can be accessed by the WSDL generator in the *symbol table* of another collaborating generator, for instance, the XML schema generator.

An overview of the subsystems of the WSDL generator is given in figure 4.2 which shows the generator and the symbol table classes within a package.

Framework architecture has been thoroughly discussed in the work of [SG96]. The overall design of an asset model compiler of a concept-oriented content management system is characterized by the usage of the framework architecture to facilitate high level control of the generators by the compiler framework.

A generator must be a *subclass* of the generic `Generator` class in the package `de.tuhh.sts.cocoma.compiler.generators`. The generic `Generator` class has four callback methods which must be implemented by a subclass, they are shown as method signatures with code excerpts.

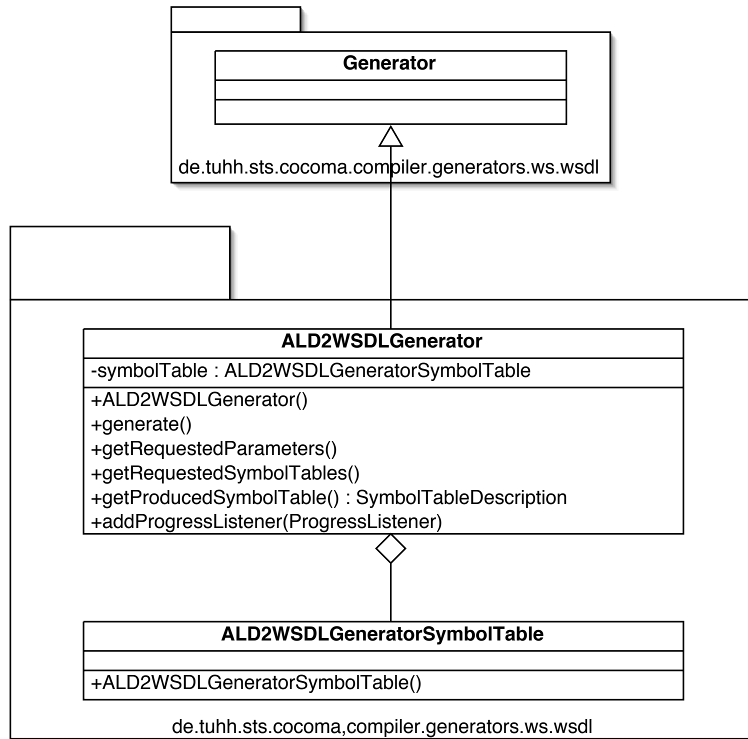


Figure 4.2: Class diagram of the WSDL generator and the symbol table

1. the *getProducedSymbolTable* method (shown in code 4.2.1) returns to the framework a copy of the *symbol table* that the generator has produced.

2. the *getRequestedSymbolTable* method (shown in code 4.2.2) requests a list of necessary symbol tables from the compiler framework. Upon invocation of this method, the compiler framework returns a list of symbol tables that are produced by other collaborating generators. The content of this list must be specified in an XML configuration file for the generator.

3. the *getRequestedParameters* method (shown in code 4.2.3) requests a list of rumtime parameters from the compiler framework. The list of requested parameters can be specified by the generator configuration file.

4. the main task of the *generate* method (shown in code 4.2.4) is to perform the actual generation chores which depend entirely on the concrete implementation of this method.

The WSDL generator must produce an internal object representation of the compiler-internal symbol information that it has produced. This object representation is written to the WSDL generator symbol table. The symbol table which is produced by the WSDL generator can be requested by other generators of the compiler framework in case that it is required to communicate the *intermediate model* of the WSDL generator with others. Figure 4.3 shows the dependencies of the generators among each other within the CCMS compiler framework using a UML diagram. In section 4.2.2 and 4.2.3 of this chapter, several issues concerning the software design of the WSDL generator will be discussed.

```
public SymbolTableDescription getProducedSymbolTable(IntermediateModel arg0);
```

Code 4.2.1: Callback method getProducedSymbolTable

4.2.2 Generator Software Architecture

Since the framework has dictated the overall requirements for the callback methods implementation of the `Generator` subclass, the design of the classes of the WSDL generator will revolve around this concept of *black box* programming towards the compiler framework.

One important task is to define a model to specify the structure of the elements of the WSDL document that is compliant with the WSDL standard specification [Con01]. This requirement is observed in the design by employing a *metamodel* which will specify the rules for converting the definitions of the module interface methods to an internal XML representation within the WSDL generator. Because this metamodel is not defined in any *symbol table* of the collaborating generators, the decision is to realize the *metamodel* using an XML instance document to store all the necessary relevant information about the methods of the module interface. This XML instance document will be mapped into an internal XML memory representation every time the generator is invoked by the compiler framework.

Another point worth mentioning concerns the versioning of the produced WSDL document itself. The W3C has released two WSDL specifications: version 1.1 [Con01] and recently a candidate recommendation version 2.0 [Con06a, Con06b, Con06c]. Because the latter specification has not yet been adopted as a standard recommendation, the web services software support is still limited at this point. The AXIS framework has introduced partial support for the version 2.0.

However, at present, the WSDL generator will be designed for the generation of WSDL document that is compliant with the widely deployed and adopted WSDL version 1.1 specification. On the other hand, in terms of a flexible design, the WSDL generator must still be conceived with the potential to expand its functionalities for generating version 2.0 as soon as the recommendation will be released on the standard track by W3C. The placeholder for this expandability in software architecture consists in the usage of the *strategy pattern* [GHJV94]. A class diagram of the *strategy pattern* is shown in figure 4.4.

4.2.3 Internal Design

Internally, the WSDL generator must maintain several important data structures. They are responsible for the maintenance of the structure of the WSDL document inside main memory. These data structures are:

- *WSDLElement*
- *WSDLAttribute*

The former is the Java in-memory representation of an XML element node and the latter represents an attribute information item of the XML infoset in the WSDL document. With the help of these elements, an arbitrarily complex structures of XML elements can be built to represent the WSDL document. A class diagram for this two important classes is given in figure 4.5 and figure 4.6.

The *symbol table* of the WSDL generator resembles any generator which is used to communicate the internal object representations with other generators. For such purpose, the content of the *symbol table* of the WSDL generator features among other things, the object representation of the XML structure of the WSDL document which is built as a hierarchical tree structure using the above mentioned elements.

```
public Collection<SymbolTableDescription>
       getRequestedSymbolTables(IntermediateModel arg0);
```

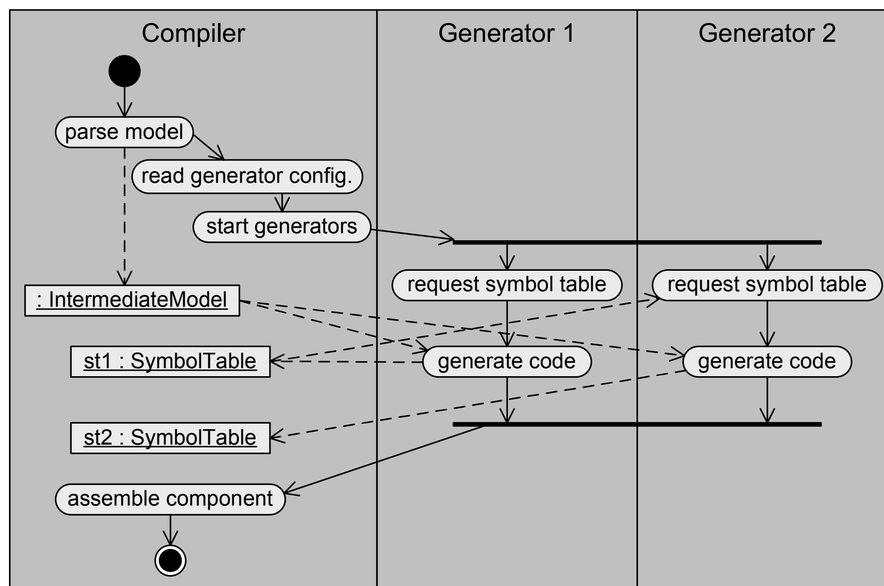
Code 4.2.2: Callback method getRequestedSymbolTable

```
public Collection<ParameterDescription<? extends Object>>
       getRequestedParameters(IntermediateModel arg0);
```

Code 4.2.3: Callback method getRequestedParameters

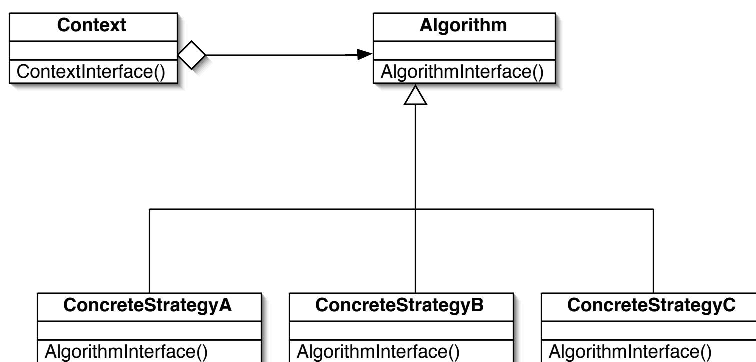
```
public SymbolTable generate(IntermediateModel arg0, SymbolTable[] arg1,
                            Map<String, ? extends Object> arg2)
       throws GeneratorException;
```

Code 4.2.4: Callback method generate



[Seh04]

Figure 4.3: Generators dependency among one another



[GHJV94]

Figure 4.4: Class diagram of the strategy pattern

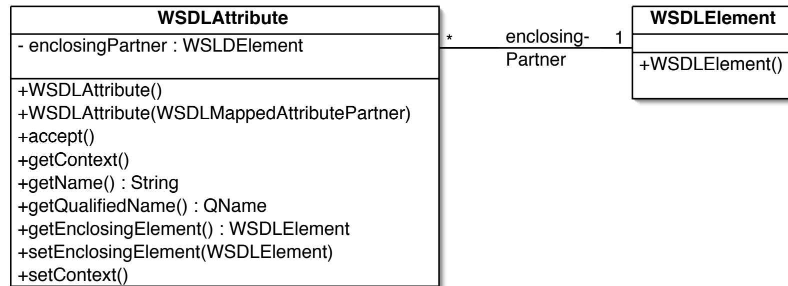


Figure 4.5: Class diagram of *WSDLAttribute* and *WSDLElement*

Beside the *symbol table* issue, the WSDL generator has the following *Java package* partition:

The `de.tuhh.sts.cocoma.compiler.generators.ws.wsdl` package contains the actual generator class and the generator symbol table.

The `de.tuhh.sts.cocoma.compiler.generators.ws.wsdl.components` package harbors the classes such as *WSDLElement* and *WSDLAttribute* which are contituent parts of the internal XML structure representation. It also contains the serialization classes to process the XML data structure for writing a WSDL document.

The `de.tuhh.sts.cocoma.compiler.generators.ws.wsdl.configurator` package contains the classes that implement the *strategy pattern* to control the generation of the proper version of WSDL document. At the same time, a *singleton* class called `ConfigurationContext` is responsible for global access to configuration information and parameters by the WSDL generator.

The `configurator.mappingrules` sub-package contains the processing classes that are responsible for the loading and reading of the mapping rules and metarules from XML configuration files containing the mapping information on the conversion of methods signatures and meta-information to the appropriate WSDL element structure.

The `de.tuhh.sts.cocoma.compiler.generators.ws.wsdl.logging` package contains a *singleton* class `WSDLGeneratorLogger` which is responsible for processing logging information for the WSDL generator.

4.3 Web Services Endpoint Design

An overview of the requirements regarding the server module are mentioned in section 3.2.2 of chapter 2. Furthermore, other requirements concerning the realization of the web-tier that are posed by the selected web technologies are elaborated in section 3.2.3 of the same chapter. Designing the web services endpoint means in principle to integrate the web-tier technologies with the asset model based CCMSs.

4.3.1 Architectural Overview

Web services calls are accepted by the communication interface of the server module of the service endpoint. Basically regarding the deployment possibilities of web services on the Java enterprise platform, the two service endpoint realizations: JSE and EJB endpoint (described in section 3.2.3) have prescribed the architecture of the service endpoint that is appropriate. The communication via web services on the server side with web services clients can use the following Java server side technologies:

- JAX-RPC servlet enabled web services endpoint,

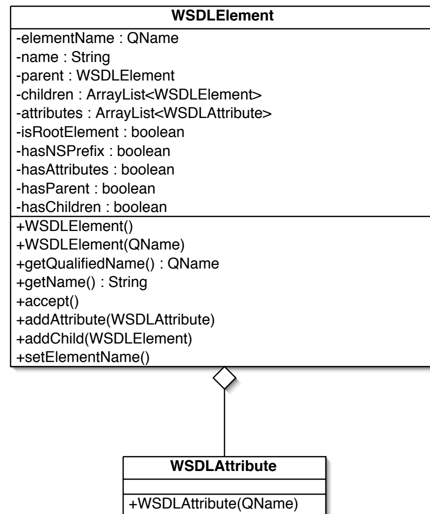


Figure 4.6: Class diagram of *WSDLElement* with *WSDLAttribute*

- stateless EJB component based web services endpoint

The design of the server side endpoint has favored the JSE[3] [MTSM03] approach for the following reasons:

- the statelessness of servlet technology is homogeneous with the web services call and processing pattern which in terms of the realization of the service endpoint is frequently carried by a stateless network transport protocol such as HTTP.

- it involves a development approach with relatively low maintenance based on generation of many software components with appropriate development tools and deployment frameworks like the Apache AXIS framework.

- the servlet approach provides container management of the entire servlet lifecycle.

- server side implementations can take advantage of the lifecycle management interface of the servlet context `ServletEndpointContext` to gain direct access to the control of the lifecycle methods such as `init()` and `destroy()` of the servlet container. Sophisticated control is thusly possible at the container level.

- good overview of configuration of the endpoint deployment since XML deployment configuration file is supported.

There are also some noteworthy disadvantages:

- no integrated transaction support,

- security is not built into the JSE processing model natively and must be implemented with extra cost,

- scalability of the endpoint in the face of a surging load might pose problem to the performance of the servlet based service endpoint.

[3] JAX-RPC compliant servlet web services endpoint

(

Package names	*de.tuhh.sts.cocoma.compiler.generators.ws.wsdl*
	de.tuhh.sts.cocoma.compiler.generators.ws.wsdl.components
	de.tuhh.sts.cocoma.compiler.generators.ws.wsdl.configurator
	de.tuhh.sts.cocoma.compiler.generators.ws.wsdl.configurator.mappingrules
	de.tuhh.sts.cocoma.compiler.generators.ws.wsdl.logging

Table 4.2: WSDL generator Java packages listing

4.3.2 The AXIS Framework

After the choice for a servlet based service endpoint has been made, the design of the endpoint begins with a survey of the Apache AXIS framework. The AXIS framework is used as a distributed SOAP framework for handling web services SOAP messages.

The Apache Software Foundation [Fou06] has produced this widely used open source web services framework which implements the SOAP protocol [Con03c] which is described in section 3.1.4. The development of web services with AXIS benefits from its abstraction of the web services development model, thereby hiding the low-level details of SOAP messages handling behind a manageable software framework.

AXIS is essentially a SOAP engine, a framework for constructing SOAP processor endpoints. The AXIS framework is implemented both in the Java and C++ programming language.

The software bundle of the AXIS framework consists of:

- a stand-alone web server,
- a servlet which is able to be integrated into a servlet engine,
- classes which extensively support the web services description language – WSDL,
- software tools which generate Java classes from WSDL or vice versa,
- a monitoring tool for viewing delivered SOAP messages transported by TCP/IP datagram packets between the participants.

On the server side, the AXIS servlet is configured to deploy web services as a web application inside a servlet container. The JAX-RPC specificiation mentioned in section 3.3.1 is implemented and supported by the AXIS framework. The JAX-RPC model tries to make web services calls as similar to RPC calls as possible. The programming model is based on mapping a WSDL concept to the corresponding Java concept. In JAX-RPC, the thing that corresponds to a portType of a WSDL document is a *service endpoint interface (SEI)*. The methods on the SEI correspond to the WSDL operations, with messages and ports being subsumed into the arguments of the SEI methods.

AXIS supports the JAX-RPC type mapping conventions which define a standardized set of mappings from XML schema types to Java types, for generating Java classes from a WSDL document. It also defines the reverse mapping, from Java types back to XML schema types. The AXIS framework defines serialization for every Java type that can be mapped from an XML schema type. The JAX-RPC type mapping conventions also include a definition for mapping JavaBean objects to XML schema types which is also supported. In the toolset of the AXIS framework, these tasks can be accomplished by the helper classes – *WSDL2Java* and *Java2WSDL*.

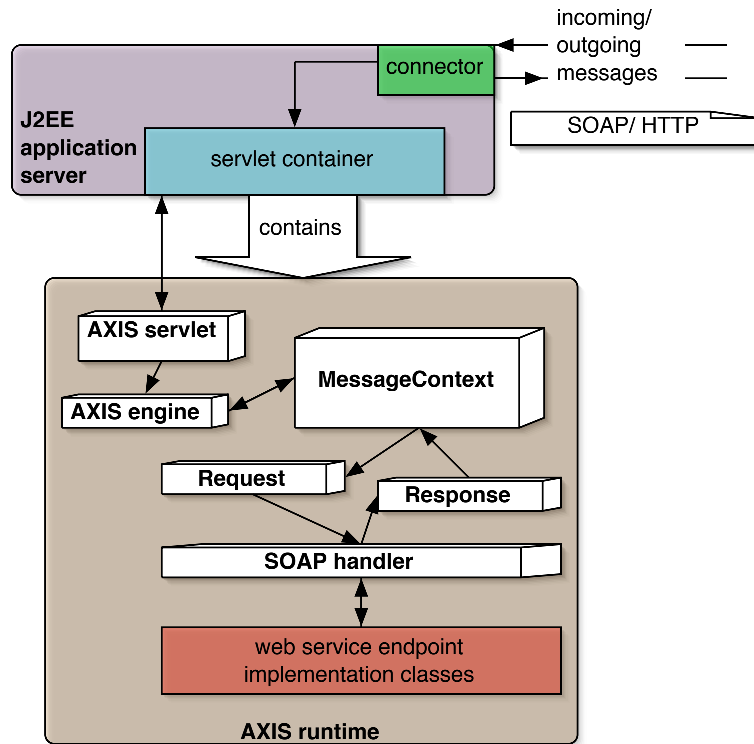


Figure 4.7: System architecture of AXIS

Applications can gain access to servlet context information by using the ServletLifeCycleInterface of the JAX-RPC specification via the defined API methods. This assumes that the JAX-RPC implementation is based on a servlet web application running inside a servlet container.

In terms of SOAP messages handling, the JAX-RPC specification provides the ability to install handlers into the SOAP message path, on both the service requestor and the service provider sides. Handlers are modularized pieces of program code which allow a program to perform operations on a SOAP message before it reaches its end destination of delivery. Handlers perform computations based on the values embedded in the headers of SOAP messages; they provide features such as message logging, encryption, or digital signatures creation. Different handlers can be configured together in the AXIS framework to form a *chain* of handlers for message transformation and the processing of SOAP messages along the message delivery path by using an XML based configuration file.

On the web services client side, there are three different client-side representations of a *port*[4]:

- stub objects generated by software tools from a WSDL document,
- a dynamic proxy-based calling interface,
- a dynamic call interface which resembles the DII interface[5].

An overview of the server side architecture of the Apache AXIS framework is shown in figure 4.7.

[4]in JAX-RPC terms, a port is a client's view of the web service endpoint
[5]DII – the CORBA Dynamic Invocation Interface

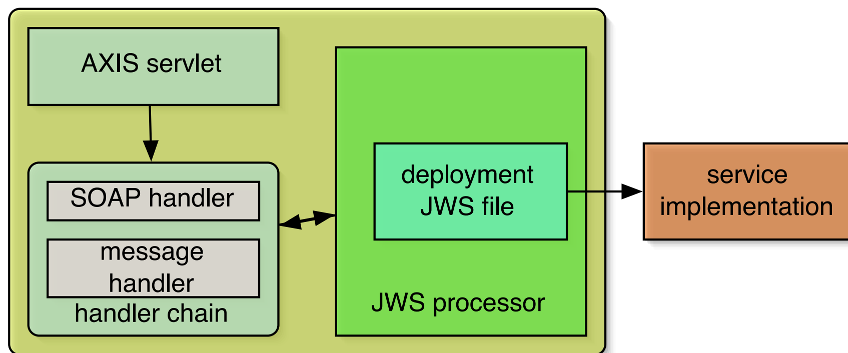


Figure 4.8: AXIS JWS deployment

The server software of the Apache AXIS framework itself contains a servlet implementation which can be run in any standard compliant servlet container. The design of the AXIS server takes advantage of the *pipeline architecture* [SG96] and the *filter pipe* messaging architecture [SG96, HWB04] in terms of the chained processing of both SOAP request and response messages by installing message processing nodes called *message handlers* along the message delivery path. The order of *message handlers* of the message chain can be configured using a framework wide configuration file called *deploy.wsdd*. A SOAP message which is processed by the AXIS server is represented by a *wrapper object* called a `MessageContext`. The `MessageContext` consists of contextual information about the server process, the request message and the response message. There are specific *default message handlers* responsible for processing the request and response messages. They are shown in figure 4.7 by name. The SOAP message *transport component* depicted is responsible for creating, sending and receiving SOAP messages over a specific concrete *transport protocol* binding such as HTTP. The SOAP *service component* depicted is responsible for creating, sending and receiving the `MessageContext` and mapping it to internal Java object representations.

The Apache AXIS servlet is the core component of the AXIS framework which processes SOAP request and response messages. The behavior of the AXIS servlet is configured by an XML configuration file called *web.xml*. The AXIS servlet reads the configuration file during initialization.

The AXIS framework provides a very simple service deployment scheme called *JWS* [Ton06] for implementing a web service by exposing concrete *JavaBean* classes to *web services auto deployment* classes called *JWS* classes and consequently renaming the classes with a *JWS* extension. Assuming that the AXIS servlet is installed and configured to handle *JWS* files, the AXIS engine automatically processes the *JWS* file when client SOAP request messages arrive. The figure 4.8 shows the concept of the *JWS* processing mechanism of AXIS.

The AXIS *JWS* scheme for implementing web services has a number of limitations. The most important one is that the mapping process from Java entities to parts of a WSDL document or vice versa is not configurable. The ability to configure the mapping of Java type entities such as interfaces or JavaBeans to XML schema entities for SOAP messages can be critical. If the use of complex Java structure is required which demands a customized mapping of the type entities, the AXIS *JWS* mechanism for auto web services deployment is not appropriate. Consequently, *JWS* web services cannot make use of Java packages.

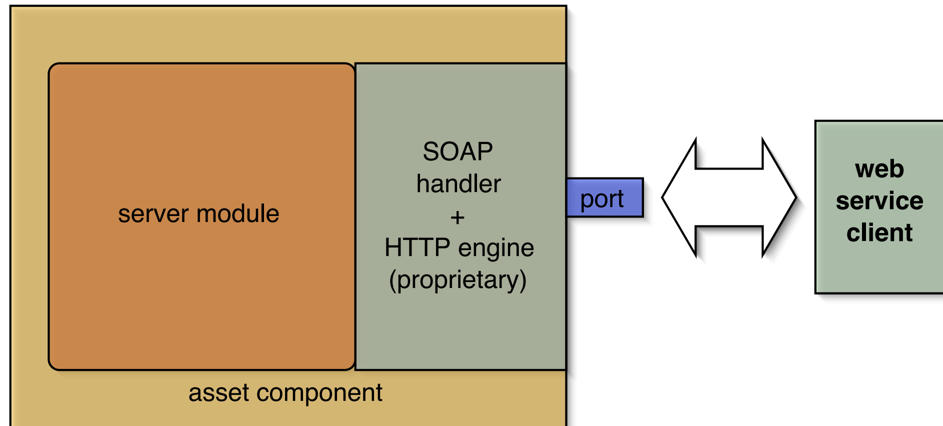


Figure 4.9: Server module design approach 1: proprietary HTTP and SOAP component

4.3.3 Web Services Endpoint Design Approaches

One of the most crucial requirements for the server module states that it must have a networking interface to listen on a specific port for HTTP requests which encapsulate the SOAP request messages. The abstract design of the server module takes into account several different approaches. In architectural terms of the server module, there are several design approaches which are summarized as follows:

1. the server module must implement the HTTP server engine and SOAP handler from scratch. For the serialization and deserialization of SOAP messages, a customized serializer must be developed. This architecture is illustrated schematically in figure 4.9; in figure 4.10 the corresponding UML deployment diagram is shown.

2. the server module externalizes the HTTP server engine and SOAP handler to a foreign component which is designed mainly for the purpose of handling requests and forwarding responses from the server module. This approach is illustrated in figure 4.11 with the corresponding UML deployment diagram shown in figure 4.12.

3. the server module coexists with the HTTP server engine and SOAP handler in parallel within a CCMS at runtime. They represents two components which are bound to a JSE servlet. The servlet serves the processing of the HTTP request and response messages on the server side. The server module resides together with the SOAP implementations within the same servlet context within a servlet container. This approach is depicted in figure 4.13 with its corresponding UML deployment diagram shown in figure 4.14.

4. the server module embeds a lightweight standard compliant servlet container within the server module itself. The server module executes within the servlet context of the embedded servlet container. This approach is illustrated in figure 4.15 with the corresponding UML deployment diagram in figure 4.16.

In the following discussion, the advantages and disadvantages of these different design approaches will be explained and the different approaches are juxtaposed with each other in order to select an appropriate design solution.

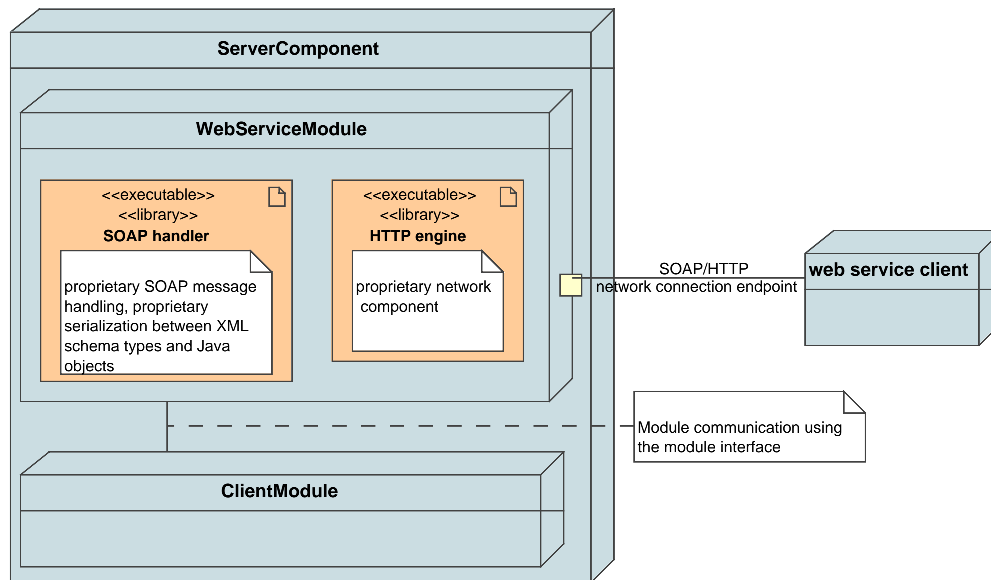


Figure 4.10: UML deployment diagram of approach 1

The first approach has the advantage of being adaptive to requirement changes and therefore it can have a higher degree of flexibility in terms of a customized design for a specific problem-solution scenario. The main disadvantage is obviously the higher cost of development. Moreover, the estimated increase in software complexity as far as a customized design and costly implementation is concerned could be prohibitive. A relatively low degree of software reuse can be guaranteed since many customized components could not be fitting into a different design; indirectly reducing the possibility of code reuse. The other side of the coin, the fact that writing a lot of customized serialization code and HTTP networking code from scratch means to re-invent the wheel most of the time. It is therefore not an effective form of practice in software development.

The second approach takes the advantage of an externalized communication unit and SOAP handling component; favoring a low-coupling design [Lar98] by distributing the processing responsibilities according to logical partitioning of tasks for the respective components. Although this approach does not suffer from the developing-from-scratch meander, the externalization of the communication unit and SOAP handling component beyond the server module means that it is also necessary to introduce a *communication interface* between the server module and the communication and SOAP handler components, this is shown schematically in figure 4.11. Moreover, it is necessary to define an *interface protocol* and implement the operations of this protocol in order to enable the two different components to interface with the server module. The main disadvantage of this design approach therefore consists in the complexity of the *interface protocol* and the unnecessary addition of heterogeneity regarding the introduction of the different *interface protocol* which is incongruent with the native asset based protocol of the *module interface*. Furthermore, as it is obvious in figure 4.11, the communication of an asset component with the J2EE application server must cross system boundary. It assumes that the asset component runs in a local instance of Java virtual machine from its own perspective and the application server instance runs on a remote instance of Java virtual machine[6]. In order to obtain a

[6]the abbreviation – VM shown in the figure stands for a Java virtual machine instance

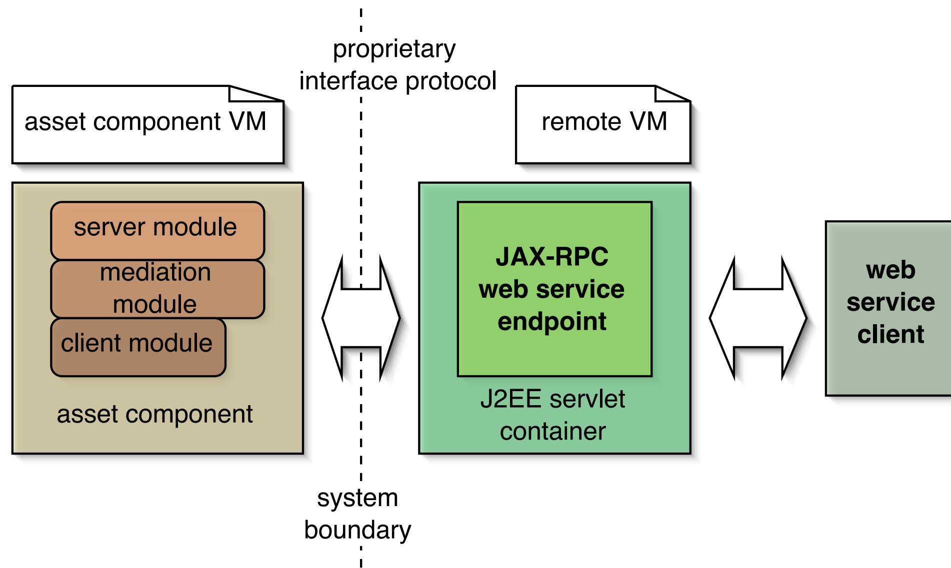


Figure 4.11: Server module design approach 2: externalization of communication component

remote object reference, there is a need for a ternary naming and lookup service such as JNDI[7] which manages the remote object references on behalf of the local instance of Java virtual machine. Yet the asset module must be accreted with another set of protocol definitions and rules which would probably over-engineer the server module interface.

The third approach resembles to a great extend the second one except that the two involved components are residing inside a servlet container instead of running on a remote Java virtual machine instance. This design takes the advantage of the directness and convenience of local object references within the same servlet runtime context. There is no need for a remote object reference naming and lookup service such as JNDI because all references are local at runtime of the server module. The server module can access object references belonging to the communication components and vice versa. On the other side, there is still the need for an *interface protocol* which characterizes the second design approach, probably bringing in extra complexity. The developer cannot waive the necessity of providing a well-defined set of protocol rules in order to facilitate the communication between the internal components. There are several similar technology solutions which were proposed in the past as widely adopted solutions on the Java enterprise platform. These server-side technologies such as the Java RMI mechanism of the J2EE platform [SM06] or the CORBA IIOP protocol [omg98] have emerged for many years.

Besides the previously mentioned server-side technologies, another possible interface protocol could even be using yet another web services interface for bridging the two components across sides of the interface. Althrough these solutions are widely adopted in server-side software development, they do not provide a simpler implementation of the *interface protocol* within the container. In worse case, they could end up in an over-engineered turmoil. Moreover, there is still the risk of introducing even more heterogeneity without alleviating complexity. The idea of providing a web services interface solution for this interface definition problem is even more far-fetched

[7] JNDI – the Java Naming and Directory Interface

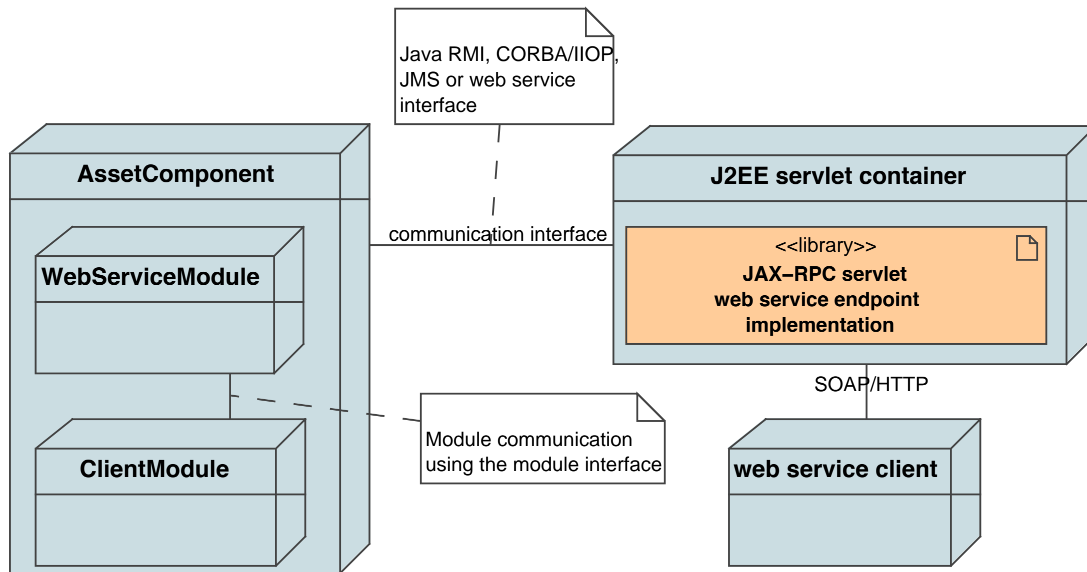


Figure 4.12: UML deployment diagram of approach 2

because the rationale for introducing that web services interface within the web container is not well proved in this context. It resembles the metaphor of cutting a loaf of bread with a saw – using the wrong tool for a task.

Finally the last approach takes advantage of an embedded servlet container. This is a promising design approach because it does not suffer from the disadvantages of the previous ones. It is characterized by the servlet container being embedded within the server module. The server module has full control of the servlet container, including starting and stopping of the HTTP server which is an integral part of the embedded container. Moreover, the memory footprint of such an embedded container is smaller comparing to the other full-blown containers such as Apache Tomcat.

4.3.4 HTTP Server and Embedded Web Container

As mentioned in section 4.3.3, the embedded container approach is favored for its ability to handle SOAP messages and provide the network functionality of an HTTP server at the same time. Therefore the server module can utilize these functionalities as if they were provided by the server module itself.

The server module is responsible for initializing the embedded container and starting the runtime container instance and the HTTP server in order to bootstrap the web services endpoint. By using the `init()` method of the *module interface*, the bootstrapping is performed by a CCMS when the mentioned method is invoked . Therefore this method must be implemented in a way that allows the bootstrapping of the embedded container from within the server module. Since the CCMS takes control of the creation and initialization of the server module by invoking the `init()` method and eventually the `start()` method, it is convenient to inline the bootstrapping code for the servlet container within the initialization method. After the module initialization is performed, the `start()` method is invoked by the CCMS to start the network interface of the container on a specific network port to listen for incoming HTTP requests.

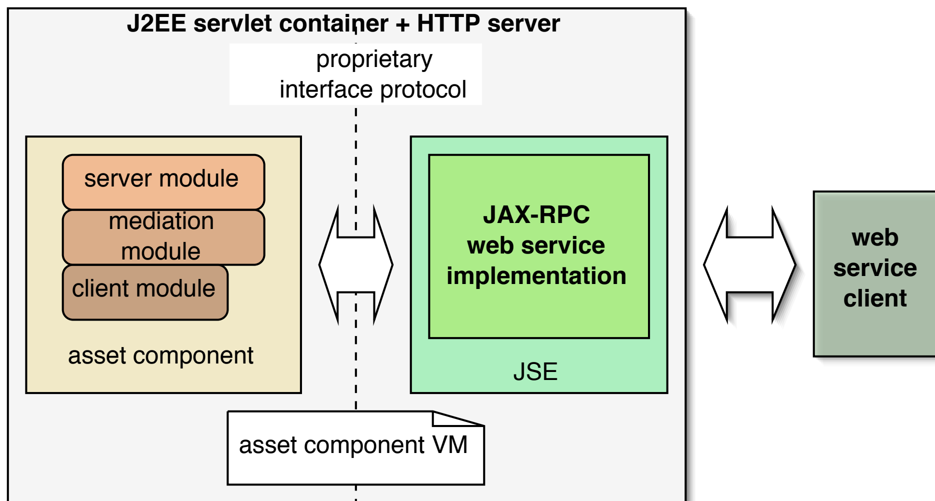


Figure 4.13: Server module design approach 3: parallel component in local context

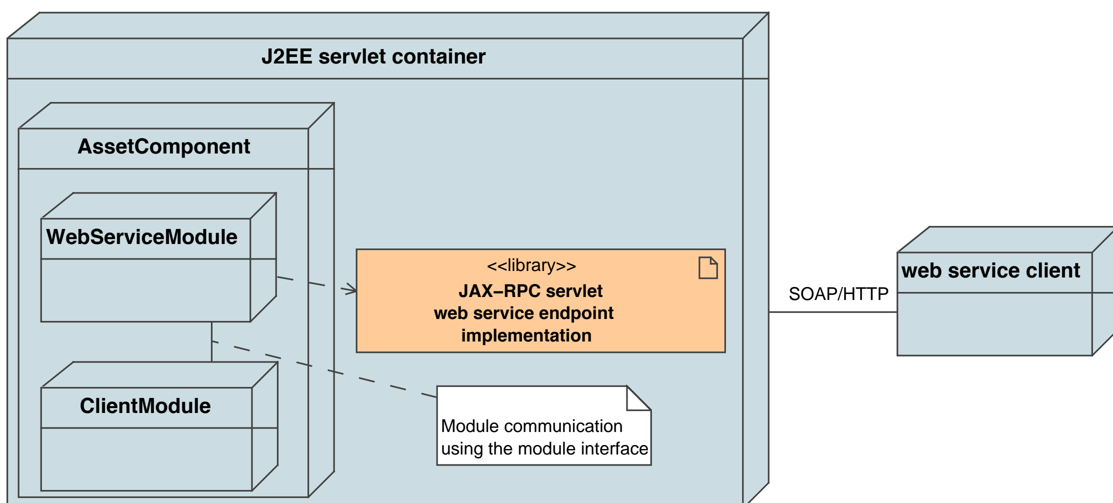


Figure 4.14: UML deployment diagram of approach 3

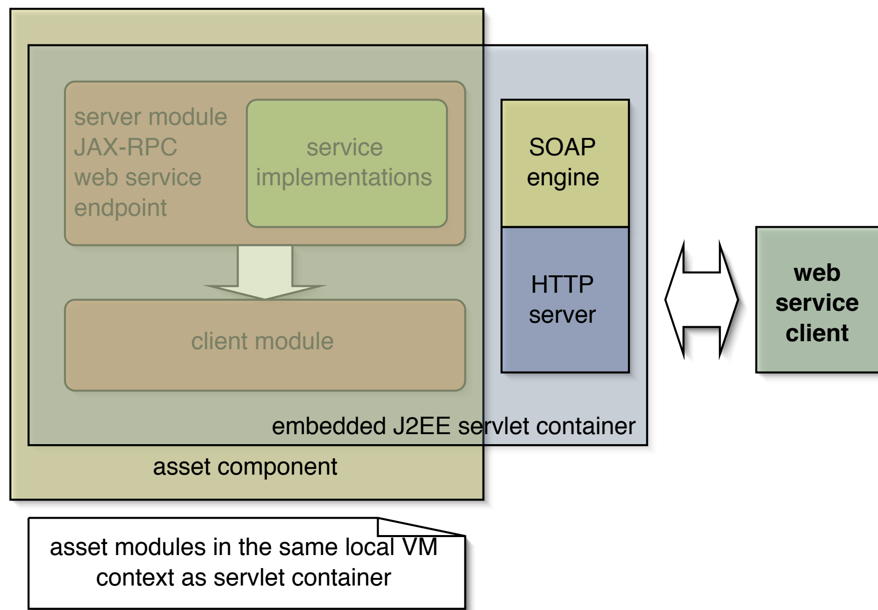


Figure 4.15: Server module design approach 4: embedded servlet container

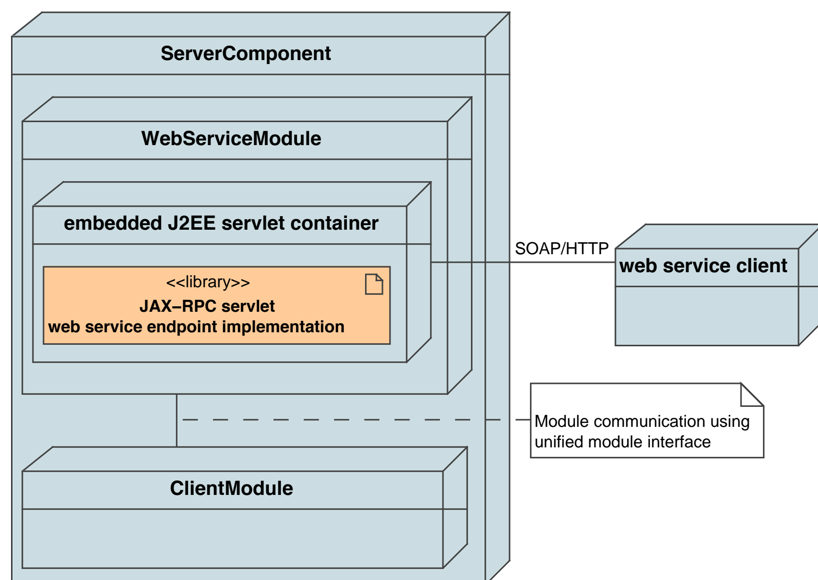


Figure 4.16: UML deployment diagram of approach 4

Chapter 5

Implementation of the WSDL Generator and Web Services Module

In this chapter some implemenation issues concerning the WSDL generator and the web services server module are discussed in several aspects. The implemenation of the WSDL generator is described in section 5.1; it is followed by descriptions of the server module implemenation and integration of the AXIS framework in section 5.2. Section 5.3 elaborates on some of the issues concerning the deployment and configuration of the web services endpoint. A descriptive account on the selected embedded servlet container – Jetty is provided in section 5.3.2.

5.1 WSDL Generator

The WSDL generator is the main piece of generator software that is implemented to fit into the backend compiler for generating a standard compliant WSDL document that describes the *module interface* of the web services server module. As such the classes implemented for the generator deserve to be mentioned. In section 5.1.1, a brief description of the implemented generator classes is given. In the following section, the exposition on the WSDL generator continues with the method implementations of the generator classes. In section 5.1.3, a brief description of the runtime configuration of the WSDL generator is provided, with a configuration example included in appendix C on page 96 of the current thesis.

5.1.1 Generator Classes

The main class of the WSDL generator is called `ALD2WSDLGenerator`. This class utilitzes the class representing the *generator symbol table* within the same package. The *symbol table* class is called `ALD2WSDLGeneratorSymbolTable`. These two classes together with the *symbol table description* class are shown in the UML class diagram in figure F.1 in appendix F.

In the UML class diagram, the main generator class contains all the defined methods with scope visibility and return type information shown in the illustration. The *symbol table* class and the `ALD2WSDLGeneratorSymbolTableDescription` class are depicted in an aggregation relationship to the main generator class.

For the purpose of XML serialization from the internal memory structure to an instance of WSDL document, a class called `WSDLSerializer` is provided to collaborate with the main generator class. This class contains the method *generateWSDL* which

is capable of traversing the internal WSDL structure in memory using an *iterator* like interface, producing a serialization for each `WSDLElement` node[1] it encounters.

The task of traversing the WSDL structure is abstracted away by using the *iterator pattern* to proceed with the traversal of the internal WSDL data structures mentioned from the top-most level to the bottom-most level that represents an instance of a WSDL document. In a recursive manner, all the `WSDLElements` in every level of the internal WSDL structure in memory are visited by the iterator interface, the method *generateWSDL* of the `WSDLSerializer` class utilizes the private method *traverseSubElement* in the class to actually visit the internal data structure. Another class called `WSDLAttribute` is used to represent attributes of the WSDL document. For consuming the XML attributes in the data structure in order to output them as well during the traversal, the private method *traverseAttribute* of the class is invoked iteratively to get the attributes together with their corresponding values serialized.

A sophisticated standardized SAX-like[2] XML streaming API called StaX is introduced by Sun Microsystem which is used for the serialization of XML to the file system. The precedent SAX interface which is bundled with the Java platform standard edition is not used because SAX is a *serial access XML parser* solely without the functionality of producing XML documents on a file system using an internal XML data structure. For the purpose of actually writing XML, traditionally, the DOM API that is already available and also implemented on the Java platform standard edition is used. But the DOM API has an infamous reputation of being problematic in terms of performance. Since a larger internal WSDL data structure can impede overall system performance if the whole XML structure must be parsed or written in a large chunk, there is no advantage in using the DOM API. Therefore the newer StaX[3] API is used. It has been introduced by Sun Microsystem for some time already and is implemented by the some third-party vendors such as BEA Systems. Sun has provided a reference implementation of the StaX API which is available through the *Java Community Process* und the title *JSR 173*. The greatest advantage compared to the SAX API is its ability to produce a serialization of the internal XML data structure in memory; a streaming XML writer is also provided in the reference implementation in order to write the instance of XML document to a file system.

With the StaX API implementation, the internal WSDL data structure can be written to a WSDL file by traversing the data structure iteratively, applying the *write* methods iteratively to the elements of the data structure. There is a large collection of *write* methods available at the developer's proposal, naming according to their functionalities respective, for instance, *writeElement* is used to write an XML element and *writeAttribute* is used for writing an attribute. It is worth mentioning that as far as serialization of the XML elements is concerned, the methods are conceived and used in pairs, that means there is always a *writeStartElement* method which corresponds to a *writeEndElement* method. Care should be taken to use the methods always pairwise; otherwise, the written XML instance document will not be well-formed.

Centralized management of configuration parameter and context information is the responsibility of a class which is called `ConfigurationContext`. It consists of data structures which store important configuration parameters and values which are read in default from configuration files on the file system. The class is implemented as a *singleton* [GHJV94] instance because it must be accessed uniquely in a global scope. The *initConfigurationParameters* method of this class is responsible for the management of a parameter map data structure which is returned by the compiler framework upon invocation of the *getRequestedParameter* method of the main generator class. The main generator class then calls the *initConfigurationParameters* method and provides the parameter map of configuration parameters and values as

[1] WSDLElement is the internal representation of an WSDL element node
[2] SAX – the Simple API for XML
[3] StaX – Streaming API for XML

a method argument; thereby initializing the WSDL generator internal configuration data structure.

The structure of the internal memory representation of a WSDL document is built by using the objects of the classes in the package *configurator.mappingrules*. These classes are listed in table 5.1 which also summarizes a selection of some other classes according to the package to which they belong.

5.1.2 Methods Implementations

The main generator class consists of the important *generate* method which must be implemented by all generators that are subclasses of the generic `Generator` class. By and large the *generate* method is the callback method that will be invoked by the compiler framework according to the schedule it has calculated for a specific generator if other backend generators must also be invoked in a row.

In order to separate different concerns on the generation task and refactor the program code in a manner that will allow better future evolutionary maintenance. The *generate* method of the main generator class is not stuffed with all possible operation code for generating the WSDL document. Instead the *generate* method of the class breaks down the tasks in certain processes which are comprehensive enough for mapping to a set of method implementations in the development process.

Software design patterns have been used in many parts of the WSDL generator, among others the *abstract interface pattern*, the *proxy pattern*, *singleton pattern*, *visitor pattern* and the *strategy pattern*. There is a general conception which states that design pattern can enhance software quality provided they are used effectively and appropriately. The design patterns mentioned represent a set of condensed and experienced software design knowledge. Therefore the advantages of using design patterns are:

- capturing software engineering experience,
- enabling comprehensive descriptions of design using UML notations,
- allowing communication of design knowledge on an abstract level,
- allowing the reuse of software design knowledge without re-inventing the wheel,
- providing a set of design vocabulary that can be mentioned conveniently.

5.1.3 Configuration of the WSDL Generator

The WSDL generator is conceived to output WSDL documents. The XML schema types which are referenced by the WSDL documents are provided by the underlying XML schema type system. The WSDL generator therefore utilizes an XML schema generator [Bos04, Bos03] for obtaining an instance of XML schema document which contains all the necessary XML schema types. The generated XML schema document is included by the generated WSDL document eventually. It is referenced by the WSDL document using an *import* element within a *wsdl:types* element within the WSDL document. The *import* directive enables the reuse of XML schema type information contained within a specific XML schema document.

Consequently, it is necessary to run the XML schema generator before the WSDL generator by using an XML configuration file to denote that execution sequence. This XML configuration file is basically an extended configuration file that resembles the configuration file for the standalone XML schema generator. An example configuration file is shown in appendix C on page 96. An instance of the generated WSDL document is available for reference purpose in appendix D on page 98.

Classes	Brief description
de.tuhh.sts.cocoma.compiler.generators.ws.wsdl	
ALD2WSDLGenerator	main generator class
ALD2WSDLGeneratorSymbolTable	generator symbol table class
de.tuhh.sts.cocoma.compiler.generators.ws.wsdl.components	
WSDLStructuralArtifact	abstract interface representing a structural element
WSDLSimpleAttribute	super class of WSDLAttribute
WSDLAttribute	in-memory representation of an attribute
WSDLElement	in-memory representation of an WSDL element
WSDLNamespace	representation of namespace information
WSDLMappedAttributePartner	enum class denoting possible attribute pair
WSDLSerializer	serializer of the in-memory XML data structure
WSDLStructuralArtifactVisitable	abstract visitor interface for traversal
WSDLStructuralArtifactVisitor	visitor for the general traversal
WSDLStreamingSerializationVisitor	specific visitor for streaming serialization traversal
de.tuhh.sts.cocoma.compiler.generators.ws.wsdl.configurator	
ConfigurationContext	centralized configuration *singleton* [GHJV94] class
WSDLVersionHandler	abstract algorithm interface for WSDL version
WSDLVersionHandlerImpl	implementation class of the interface handler
WSDLStructureAssemblyAlgorithms	abstract *strategy pattern* [GHJV94] *algorithm interface*
WSDLStructureAssemblyStrategy	abstract class implementing the *algorithm interface*
WSDLStructureAssemblyContext	abstract *strategy pattern* [GHJV94] *context interface*
WSDLStructureAssemblyManager	concrete *singleton* class implementing the *strategy pattern context interface* [GHJV94] for assembling the WSDL data structure internally
WSDLVersion1Strategy	concrete class with the *build* method to assemble version 1.1 WSDL structure by extending the abstract class implementing the *algorithm interface*
WSDLVersion2Strategy	concrete placeholder class for version 2.0 enabling future expandibility
de.tuhh.sts.cocoma.compiler.generators.ws.wsdl.mappingrules	
WSDLConfigurationMappingRulesManager	*singleton* class managing method mapping process
WSDLConfiguration	in-memory representation of mapping rule data structure
WSDLMethodMapping	class collaborating closely with manager class
MappingRule	in-memory representation of an mapping rule entry
de.tuhh.sts.cocoma.compiler.generators.ws.wsdl.logging	
WSDLGeneratorLogger	*singleton* class centralizing logging facility for the WSDL generator

Table 5.1: Summary of classes of the WSDL generator

5.2 AXIS Framework Integration

The choice a design approach for the web services server module as described in section 4.3.3 has favored the embedded servlet container approach. The Apache AXIS framework has been described exhaustively in section 4.3.2 and in many other related books. For the reason of developing the web services sophisticatedly for deployment; regarding to the versatility and configurability of the AXIS framework, it is used as the appropriate SOAP framework to provide an integration solution with CCMSs to implement a web services JSE.

5.2.1 Setup and Configuration

JAX-RPC Web Services Implementation Models

The JAX-RPC specification defines two approaches for implementing web services: a simple model of Java servlet endpoint using the Java RMI programming model or an EJB model. These differences in endpoint design has been elaborated previously already. The following description will concentrate on the implementation of a servlet based web services endpoint. When using a Java servlet, the web services implementations are run together with the servlet within a servlet container. The web services utilize a deployment descriptor file called *web.xml* to configure the details of web services deployment. This includes the configuration of associated specific SOAP *message handlers* components as well as the type mapping information for the various Java-to-XML serialization within the AXIS framework.

In general, the deployment model for J2EE based web services involves taking the following steps:

- deploy the implementation components – the web services implementations must have a standard component entry in a web application archive. This can be a `<servlet>` entry in the *web.xml* deployment descriptor file as shown in code 5.2.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
    <servlet>
        <servlet-name>AxisServlet</servlet-name>
        <display-name>Axis Servlet</display-name>
        <servlet-class>
            org.apache.axis.transport.http.AxisServlet
        </servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>AxisServlet</servlet-name>
        <url-pattern>/servlet/AxisServlet</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>AxisServlet</servlet-name>
        <url-pattern>/services/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Code 5.2.1: Web service deployment descriptor file *web.xml*

- create a web service deployment descriptor – it takes the form of an XML configuration file called *webservices.xml* which is often included in the web archive of a web application. An excerpt of this configuration file is given in code example 5.2.2. It shows all the deployed web services within a specific web application

archive as a list; by linking each deployed web service to a particular component and specifying the corresponding WSDL file for this web service together with the Java-to-XML mapping files. The *webservices.xml* deployment descriptor file augments the chief deployment descriptor file *web.xml*. It enumerates the web services in the endpoint and references the servlet components which are associated with the web services. The *webservices.xml* file is placed in the `WEB-INF` directory of a web application archive alongside with the *web.xml* deployment descriptor file. It is worth mentioning that the newer versions of AXIS have adopted the same XML configuration file approach but with a centralized file called *server-config.wsdd* using similiar configuration directives and contents.

```
<?xml version="1.0" encoding="UTF-8"?>
<webservices xmlns="urn:de.tuhh.sts.cocoma.generic"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             version="1.1">
    <webservice-description>
        <webservice-description-name>
            CocomaGenericSModuleService
        </webservice-description-name>
        <wsdl-file>WEB-INF/wsdl/cocoma_s_module_interface.wsdl</wsdl-file>
        <jaxrpc-mapping-file>WEB-INF/cocoma-mapping.xml</jaxrpc-mapping-file>
        <port-component>
            <port-component-name>
                CocomaGenericSModuleServicePort
            </port-component-name>
            <wsdl-port>CocomaGenericSModuleServicePort</wsdl-port>
            <service-endpoint-interface>
                de.tuhh.sts.cocoma.generic.CocomaGenericSModuleService
            </service-endpoint-interface>
            <service-impl-bean>
                <servlet-link>CocomaGenericSModuleBindingImpl</servlet-link>
            </service-impl-bean>
        </port-component>
    </webservice-description>
</webservices>
```

Code 5.2.2: Web service deployment descriptor *webservices.xml*

- generate a WSDL document descriptor for the web service – each web service must be bundled with a WSDL descriptor within the web application archive; the *webservices.xml* deployment descriptor references the WSDL file for each web service. The task of the WSDL generator is to generate the required WSDL document. The *webservices.xml* file requires a WSDL file for each web service at deploy time because the J2EE specification does not assume that every Java web services implementation would be capable of generating a WSDL descriptor automatically at runtime. In some cases a specific WSDL document must be produced manually.

- generate a JAX-RPC mapping file for each web service to deploy – each web service must have a mapping file which specifies how various Java object entities in the implementation are to be mapped into XML entities, i.e. concrete instances of the XML schema types in an XML schema. This includes mapping Java packages to XML namespaces, Java methods to WSDL operations, Java objects to XML schema types and method arguments to the correct WSDL messages. The J2EE specification requires a mapping file at deployment time. It is shown in code 5.2.3.

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
            xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
  <service name="CocomaGenericSModuleService" provider="java:RPC">
    <!-- the implementing class for the service -->
    <parameter name="wsdlTargetNamespace"
               value="urn:de.tuhh.sts.cocoma.generic"/>
    <parameter name="className"
               value="CocomaGenericSModuleBindingImpl"/>
    <parameter name="wsdlServiceElement"
               value="CocomaGenericSModuleService"/>
    <parameter name="wsdlPortType"
               value="CocomaGenericSModulePortType"/>
    <parameter name="wsdlServicePort"
               value="CocomaGenericSModuleServicePort"/>
    <parameter name="schemaUnqualified"
               value="urn:de.tuhh.sts.cocoma.generic"/>

    <!-- methods that are exposed as operations -->
    <parameter name="allowedMethods" value="*"/>
    <parameter name="schemaQualified"
               value="urn:de.tuhh.sts.cocoma.generic.smodule"/>
    <parameter name="typeMappingVersion" value="1.2"/>

    <!-- a list of mapped types that are handled by AXIS -->
    <typeMapping
      deserializer="org.apache.axis.encoding.ser.BeanDeserializerFactory"
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      qname="ns30:Asset"
      serializer="org.apache.axis.encoding.ser.BeanSerializerFactory"
      type="java:generic.cocoma.sts.tuhh.de.Asset"
      xmlns:ns30="urn:de.tuhh.sts.cocoma.generic"/>
    <!-- some types are similar and are consequently masked out -->
    <typeMapping
      deserializer="org.apache.axis.encoding.ser.BeanDeserializerFactory"
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      qname="ns1:Kommentar"
      serializer="org.apache.axis.encoding.ser.BeanSerializerFactory"
      type="java:smodule.generic.cocoma.sts.tuhh.de.Kommentar"
      xmlns:ns1="urn:de.tuhh.sts.cocoma.generic.smodule"/>
    <typeMapping
      deserializer="org.apache.axis.encoding.ser.BeanDeserializerFactory"
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      qname="ns35:Fund"
      serializer="org.apache.axis.encoding.ser.BeanSerializerFactory"
      type="java:smodule.generic.cocoma.sts.tuhh.de.Fund"
      xmlns:ns35="urn:de.tuhh.sts.cocoma.generic.smodule"/>
    <!-- etc. -->
    </service>
</deployment>
```

Code 5.2.3: Mapping descriptor *cocoma-mapping.xml*

5.2.2 Generation of Service Implementations from WSDL

Upon obtaining the WSDL document using the WSDL generator, the AXIS framework can be used to generate the necessary server-side implementation skeleton code, type mapping code as well as the deployment and undeployment descriptors. This can be achieved in two ways: by running the *WSDL2Java* class bundled with the AXIS framework on the command line or by using an `ant` compile target which to generate the required software artifacts. The command line argument for running the *WSDL2Java* class is shown in code 5.2.4.

```
java org.apache.axis.wsdl.WSDL2Java -o generated --deployScope Request
     --noWrapped --skeletonDeploy true --server-side --buildFile
     --typeMappingVersion 1.1 -H wsdl/cocoma_s_module_interface.wsdl
```

Code 5.2.4: WSDL2Java class invocation arguments

The generation tool outputs the skeleton code of a JAX-RPC compliant service implementation class. This class can be implemented by using *delegation* method calls via the proxy class `ProxyAgent` to obtain a reference to the server module; then invoking the corresponding methods of the server module interface respectively. The chain of method calls will continue because the server module itself delegates these calls further to invoke the corresponding module methods of a base module. The convenient side effect of the generation process mentioned above is that it also generates the necessary XML-Java object serizalization code and the necessary serializer implementations.

5.2.3 Service Deployment in J2EE Servlet Container

The AXIS framework uses an XML configuration file called *server-config.wsdd* to configure the AXIS servlet for web services deployment. This configuration file specifies the name of the service and the Java components which implement the web service. It also includes necessary mapping information for the bidirectional Java-XML serialization. The *WSDL2Java* generation tool mentioned previously can generate an extra pair of configuration descriptors called *deploy.wsdd* and *undeploy.wsdd* which are used by the AXIS framework to deploy and undeploy the web services from the AXIS servlet dynamically.

The dynamic deployment or undeployment of web services in the AXIS servlet is as shown in code 5.2.5. One deployment detail that is not included in the mentioned AXIS *wsdd* descriptor files is the mapping of XML namespaces to Java package names; however it can be handled using a separate namespace mapping configuration file.

5.3 Server Module

The server module must be implemented correctly so that its interface definitions are compliant with the *module interface* definition of a CCMS module. This premise must be observed strictly when embedding a servlet container inside the module. Therefore the bootstrapping of the servlet container must be provided inside the initialization method of the server module instead of adding an extra method for this purpose to the *module interface*. The implementation issues of the server module are described in the following sections.

5.3.1 Server Module Configuration

In order to start the CCMS modules, an XML configuration file is used to define the hierarchy of modules that are initialized by the system. This file is named *cc.xml*

```
java org.apache.axis.client.AdminClient
     -lhttp://ws.cocoma.de:8080/axis/servlet/AxisServlet
     WebContent/WEB-INF/deploy.wsdd

java org.apache.axis.client.AdminClient
     -lhttp://ws.cocoma.de:8080/axis/servlet/AxisServlet
     WebContent/WEB-INF/undeploy.wsdd
```

Code 5.2.5: Service deployment arguments

and is shown for an example configuration scenario in code example 5.3.1. In this configuration file, there is the web services server module mapped to the corresponding module implementation class given in the element *server-module*. It is configured to delegate it web services operation calls to a *base module*[4] which is given using the element *client-module*.

The main task of the server module consists in the delegation of the request calls to its base module and the forwarding of the base module responses to the JAX-RPC web services implementation classes generated by the AXIS framework.

5.3.2 Configuration of Embedded Servlet Container — Jetty

The embedded Jetty servlet container must be configured and prepared for deploying the AXIS servlet before the server module is initialized and started by a CCMS. The server module class together with all its dependant classes and the AXIS generated implementation classes such as the serialization classes and the serializers must be compiled before the deployment. In case that a web application archive file is not used to deploy the service, these mentioned Java classes must be placed inside the `classes` folder which resides in the `webapps/axis/WEB-INF/` web application directory of the servlet container.

5.3.3 Module Initialization and Modification

Both the server module and the the base module will be initialized and started by a CCMS at runtime. It invokes the `init()` method of the modules respectively. Eventually, the system starts the lifecycle management for the modules by invoking the `start()` method.

The Jetty embedded servlet container is started concurrently when a CCMS invokes the `start()` method of the server module because the bootstrapping code is inlined in the method implementation. The following code excerpt shows the necessary modification of the `start()` method.

During implementation of the web service classes, it turns out that it is necessary for the method implementations in the AXIS generated classes to have code which requires access to object references local to the server module. Because the server-side method implementation code must invoke the corresponding module methods of the server module in order to delegate calls between the two classes. The solution is to find a mechanism which does not violate the principle of class encapsulation of an object-oriented design, for instance, to render the server module class completely public for the server-side method implementations. On the other hand, the actual code of object references should be decoupled from the detail of the object access mechanism instead. Therefore the code should not be inserted directly into the method implementations of both classes.

In order to enable the AXIS service implementation classes to communicate with the server module implementation class by seemlessly and conveniently obtaining the

[4]A base module is also known as a client module

```
<?xml version="1.0"?>
<cc:component name="component-driver"
    xmlns:cc="http://sts.tuhh.de/cocoma/component-config.xsd">
 <cc:server-module name="ws-server-module"
  class="de.tuhh.sts.cocoma.generic.servermodule.impl.ServerModuleWSModImpl">
    <cc:base-module ref="base"/>
 </cc:server-module>
 <cc:client-module name="base"
  class="de.tuhh.sts.cocoma.generic.basemodule.impl.ClientModuleBaseModImpl">
    </cc:client-module>
</cc:component>
```

Code 5.3.1: Component deployment descriptor *cc.xml* for the runtime configuration of a CCMS

```
public class ServerModuleWSModImpl implements ServerModule {

  //
  // other methods are masked out for brevity
  //

  public void start() throws ModuleException {
    Server server = new Server();
    SocketListener listener = new SocketListener();
    listener.setPort(8080);
    server.addListener(listener);

    // graceful shutdown
    server.setStopAtShutdown(true);
    server.setStopGracefully(true);

    // definition of the web application archive directory
    try {
      server.addWebApplications("C:/cocoma/webapps");
    } catch (IOException e) {
      // IO exception
      // throw a module exception to signal an exceptional event
      throw new ModuleException(e);
    }
    try {
      server.start();
    } catch (Exception e) {
      // throw a module exception
      throw new ModuleException(e);
    }
    try {
      server.join();
    } catch (InterruptedException e) {
      // throw a module exception
      throw new ModuleException(e);
    }

  }

// code bootstrapping embedded Jetty
}
```

Code 5.3.2: Bootstrapping Jetty servlet container in the server module

necessary object references at runtime, the *proxy* design pattern is used to provide a solution to this issue. The code of obtaining local object references is refactored into a *singleton* class named `ProxyAgent`, with the chosen name reflecting the usage of the *proxy* design pattern. This class provides a seemless layer to manage object references on behalf of its client classes. It relays the object references calls from the AXIS service implementation classes to the server module and vice versa, so that the server-side of the service endpoint can be connected to the server module in order to allow the AXIS classes to invoke methods in the server module interface. The *singleton* class can therefore be accessed by both the AXIS framework and a CCMS. The implementation of this proxy class is shown in appendix E on page 105.

Within the `ProxyAgent` class the `register()` method is used by the server module class `ServerModuleWSModImpl` to register itself as the server module in the proxy class. The proxy class remembers the reference of the server module so that the AXIS implementation classes can obtain a reference to the server module by accessing this *singleton* instance in a CCMS at runtime. The call chain is eventually accomplished by invoking the `lookup()` method using the module name as an argument. The reference is lookuped and returned to the service implemenation classes so that the interface methods of the server module can be invoked eventually.

It is important for the server module to register itself with the proxy class. This is best done when a CCMS initializes the module by invoking its `init()` method. Code excerpt 5.3.3 shows the modified `init()` method for registration.

```
public class ServerModuleWSModImpl implements ServerModule {

  //
  // other methods are masked out
  //
  public void init(Component arg0, String arg1, Map arg2)
              throws ModuleException {

    // init the name proxy and register the module and the component
    ProxyAgent.getInstance().register("ws-server-module", this);

    ProxyAgent.getInstance().registerComponent("parentcomponent", arg0);
  }
  // module registration
}
```

Code 5.3.3: Server module registers with the *singleton* proxy class

An extra advantage of using an embedded servlet container consists in the fact that all object references are local to the current Java virtual machine instance; it facilitates convenient message exchange between objects at runtime without having to rely on a remote object access mechanism such as JNDI.

Chapter 6

Summary and Outlook

As web services continue to evolve and gain momentum as an important technology in the realm of application integration. Regarding the advantages of adopting web services, it is advantageous for a CCMS to use this technology as an interface technology to communicate with heterogeneous systems, providing the asset management capabilities via CRUD operations to a variety of other systems which are otherwise not capable of interacting with a CCMS over the network.

6.1 Conclusion and Assessments

The analysis of the requirements for a server module that is deployed inside a servlet container has shown that the web services provision of the asset system via a web service server module is possible. While the development process of the web services endpoint can begin either with a WSDL document that describes the module interface to be exposed as web services operations or with the implementations of the module server classes and generate the WSDL document afterwards, the actual design has adopted the WSDL document approach. The WSDL document can be generated by an WSDL generator which is run inside the CCMS compiler by collaborating with the XML schema generator to produce the required WSDL document.

On the server-side the embedded servlet container Jetty has enabled the provision of an HTTP server with a standard compliant servlet container that is embedded within the server module. With the Apache AXIS framework, the web services endpoint can be built by generating the server-side skeleton classes which are directly mapped from the constituent parts of a WSDL document by the AXIS framework. Because the module interface method definitions are invariant, the generated skeleton classes can be replaced by a set of invariant implementation classes of the server module interface. The server module implementation can collaborate with the AXIS framework for fulfilling the communication needs, serialization to and from XML schema types and Java objects and processing SOAP messages. To achieve a certain degree of low-coupling of the two frameworks in terms of object references which must be known to the runtime code of both the server module and the AXIS framework, a proxy singleton class can be used to manage the references of remote objects. It is not necessary to generate the server-side implementations since the same implementations can be reused once they are readily implemented.

The client side can access the generated WSDL document to discover the web services interface definition, therefore the generation of client-side code is unproblematic because the WSDL document can be downloaded from the server module over the network by accessing the corresponding URL of the AXIS servlet.

At present, since the development of the web services server module is in an initial state, development endeavors in a similar direction like the web services server module

for a CCMS have not been undertaken yet. However there are other development efforts to provide a server module using other server-side J2EE technologies at STS.

6.2 Outlook

Since there is a lack of precedent works concerning the integration of CCMSs with a Java web services framework such as AXIS, advances in both fields will be expected in the near future. Overlooking the current development effort, it has revealed some more advanced issues which require closer attention and research for future development. These issues are:

- transaction support for the server module,
- caching of query results,
- security model and access control model.

Transaction Support

In case the server module has to handle concurrent client requests with operations that implicate transaction processing, for instance, a client requests the server module to *modify* an instance of asset by providing an *id* of an asset instance to a CCMS; it then *deletes* this instance while another client wants to read from the same asset instance concurrently by requesting the server to perform a *lookfor* operation on that instance. Such classic transaction processing scenario is often the real day-to-day operation scenario that a CCMS has to cope with.

While clients can communicate through the web services operations with the server module and forget about the transaction processing concerns of the CRUD operations involved, the web services server module should be able to deal with these transaction operations on behalf of the web services clients anyway. This means that the server module must know how to come to agreement terms on the transaction protocol used, handling client coordinations, and issuing locks on critical resources appropriately. Some organizations such as the W3C and OASIS have been working on web services transaction standards. Several standard specifications for web services atomic transaction and web services coordination have been released already. Since these standards are still not yet very widely available as standard software or web services frameworks in a stable state, over time the transaction processing functionalities will remain a research focus of interest to be taken into account by programmers. The near future will see more programmers working to provide more stable implementations of web services frameworks which would integrate and support better web services transaction operations.

In principle, it can be suggested that transaction support of the server module can be built into the software using two different approaches:

1. build a CRUD operations command interpreter to support transaction processing of more complex CRUD operations or operation sequences,
2. build a transaction workflow scheduler which utilizes the *transaction bracketing* [BN97] technique to save sequence of transaction operations in the server module before these are scheduled for execution by the module.

In the first approach, the server module could be extended by developing a *CRUD operations command interpreter* to support more complex sequences transaction of CRUD operations. The idea behind the command interpreter is underlied by the possible syntax extension of some CCMS CRUD operations which allow the nesting

of one operation inside the other, or putting that in other words, one CRUD operation takes another CRUD operation as parameter instead of taking only simple asset instances or expressions which evaluate to assets as parameters. One of these kind of CRUD operation examples is already shown in code example 2.1.6 of chapter 2. In that example the *delete* operation takes as parameter an expression which itself represents another CRUD operation – the *lookfor* operation. Because the asset language does not set limit for such extended syntax in the language usage, CRUD operations takes on a transactional dimension by using commands such as those similar to the example of the mentioned code, it is necessary to lend the server module with the capability of analyzing, parsing and scheduling these operations in a manner so that more complex transactions can be broken down into manageable atomic CRUD operations or sequences of operations which are then scheduled by a scheduler for execution. One interesting point to mention here is that whether the scheduler must have knowledge of client context information for the identification and association of transaction requests with their initiators. It turns out that in a distributed environment, this identification information is needed and must be taken into account in the design of the scheduler and workflow controller.

The second approach basically states that a workflow controller, whether it is implemented as a server module extension or as a stand-alone component which cooperates with the server module, it must *bracket* the transactional operations before it actually invoke the correct CRUD operations on behalf of the requesters. It means that the controller must implement buffers for storing the operation requests so that it can remember the entire sequence of operations within the transaction bracket before executing the individual atomic transaction operations. Traditionally, this is accomplished by issuing a *begin of transaction* – *BOT* command before the first operation of the entire transaction sequence and an *end of transaction* – *EOT* command after the last atomic operation is read into the buffer. The task of the workflow controller is to start the transaction and let the scheduler coordinate the atomic operations. The controller then waits for the completion of the operations; finally arbitrating whether a *commit* operation will be executed if all the precedent atomic operations are successful or to execute the *abort* operation in case one operation fails to complete. It implies that the workflow controller must *remember* the return status of the operations in the entire sequence as soon as it starts the transaction. It must also remember the intermediate states between two atomic operations using logging techniques in order to undo the changes or compensate the effects of the precedent operation in case a *rollback* operation is requested. Furthermore, since different CCMSs can be distributed widely over the network, it is therefore necessary for a server module to support the 2PC protocol[1] because multiple distributed transactions must either all commit at the same time in case of successful operations or all abort and release resource in case of failure. The goal of using the 2PC protocol is to ensure that a transaction can meet this requirement. For more details on the 2PC protocol implementation see [BN97].

Caching Support

A sophisticated caching mechanism can increase the overall system performance of query operations. The performance of the system perceived by a client can be enhanced if the server module would cache assets it gets from the base module on behalf of a client.

The server module could introduce an appropriate cache replacement algorithm to invalidate older cache entries base on some specific criterias, for instance, using a timestamp to track the time elapsed since the the last create or modify operation on an asset in the cache entry which the server module gets from the base module. The responsibility to invalidate older cache entries in the server local cache belongs

[1] 2PC – the two phase commit protocol

to the server module. In order to enforce the replacement algorithm strictly, it means that the server module must take asset *state* or lifecycle information into account to keep track of the validity of the cache entries. On one side, caching will increase overall system response rate; however this approach also introduces a greater amount of processing overhead for the server module. Occasionally, the responsibility to invalidate older cache entries could be shifted from the server module to a web services client by forcing the client-side to keep a local client cache. By and large caching is definitely crucial when it comes to enhancing server module performance.

Security and Access Control Model

Information systems are accessed by users continuously, The ubiquitous security risks that malicious users impose on the web services server module such as unauthorized access or malicious attacks have threatened the safety of the information system as a whole. The necessity of a sophisticated security model for the exposed web services endpoint is therefore very obvious.

The appropriate security model for the web services server module should meet the following requiements:

- authentication and authorization of web services clients,
- transport security with encryption.

Some web services security frameworks such as the WS-Security [ftAoSISO04] and WS-Trust [ftAoSISO06b] framework implementations can be integrated with the server module to add a security and authentication layer to the web services endpoint.

For client authentication, the WS-Security standard introduces a *credentials header* that contains a set of credentials information encapsulated in the header portion of the exchanged SOAP messages. Inclusion of multiple credentials can be used for multiple authentication sessions. Digital certificates can be used to authenticate a web service client based on its claimed identity.

The aspect of transport security begins with securing the communication channel between the web services endpoint and web services clients via cryptographic technologies. A web server or application server is generally used to deploy a web services endpoint; consequently an adequate security model can use the built-in security features in a web server or an application server. The fact that many popular web server and application server platforms can support encryption mechanisms for HTTP using secured communication channel means that the communication can be adequately secured between web services participants. Encrypted communication channels using the TLS[2] technology are often used for this purpose. TLS has become the de facto standard to encrypt data transmitted between HTTP requesters and HTTP servers. It operates at the network session layer and provides point-to-point message confidentiality, unidirectional or bidirectional HTTPS authentication. Security tokens like *Digital certificates* and the *public key cryptography* technology play the central role in this security model. The server authentication operations and optionally client-side authentication can create a secured HTTP session where all network traffic between the two parties is encrypted to secure message confidentiality and integrity. TLS can be used as a baseline web services communication security mechanism. However, a more sophisticated way to ensure peer-to-peer security should be a technology which incorporates security at the application level. But incorporating security into an application is often a complicated task. It turns out that by adding a message-level security layer to the SOAP messages is a good solution. This approach is adopted by the recently released web services security standards such as WS-Security.

An interesting approach to manage client access control is to map user identities of the web services clients themselves into a representation of asset models containing

[2]TLS – Transport Layer Security, the successor of SSL

a set of user asset classes. The users identities become literally manageable asset instances inside a CCMS. By modeling the resources to which access control must be enforced in connection with the user asset classes, for instance, by creating relationships of the user asset classes to secured resources within a security model, sophisticated access control can be enforced by providing access control *constraints* on the protected asset resources. A CCMS could manage a set of asset control lists based on this principle mentioned to build an effective access control model.

Appendix A

XML Schema of Generic Types

```
1   <?xml version="1.0" encoding="utf-8"?>
2   <xsd:schema targetNamespace="urn:de.tuhh.sts.cocoma.generic"
3               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
4               xmlns:tns="urn:de.tuhh.sts.cocoma.generic"
5               xmlns:generic="urn:de.tuhh.sts.cocoma.generic">
6           <xsd:element name="AbstractAsset" type="generic:AbstractAsset"/>
7           <xsd:element name="Asset" type="generic:Asset"/>
8           <xsd:element name="AssetClass" type="generic:AssetClass"/>
9           <xsd:element name="AssetList" type="generic:AssetList"/>
10          <xsd:element name="AssetIterator" type="generic:AssetIterator"/>
11          <xsd:element name="QueryConstraint" type="generic:QueryConstraint"/>
12          <xsd:element name="CharacteristicQueryConstraint"
13                       type="generic:CharacteristicQueryConstraint"/>
14          <xsd:element name="ToManyRelationshipQueryConstraint"
15                       type="generic:ToManyRelationshipQueryConstraint"/>
16          <xsd:element name="ToOneRelationshipQueryConstraint"
17                       type="generic:ToOneRelationshipQueryConstraint"/>
18          <xsd:element name="MemberInitialization"
19                       type="generic:MemberInitialization"/>
20          <xsd:element name="AttributeInitialization"
21                       type="generic:AttributeInitialization"/>
22          <xsd:element name="CharacteristicInitialization"
23                       type="generic:CharacteristicInitialization"/>
24          <xsd:element name="ContentInitialization"
25                       type="generic:ContentInitialization"/>
26          <xsd:element name="RelationshipInitialization"
27                       type="generic:RelationshipInitialization"/>
28          <xsd:element name="ID" type="generic:ID"/>
29          <xsd:element name="NoAsset" type="generic:NoAsset"/>
30          <xsd:element name="ConstraintDescription.ComparisonOperator"
31                       type="generic:ComparisonOperator"/>
32          <xsd:complexType name="ID">
33                  <xsd:sequence>
34                          <xsd:element name="componentName" type="xsd:string"/>
35                          <xsd:element name="moduleName" type="xsd:string"/>
36                          <xsd:element name="internalID" type="xsd:string"/>
```

```
37                     <xsd:element name="hashCode" type="xsd:int"/>
38             </xsd:sequence>
39     </xsd:complexType>
40     <xsd:complexType name="NoAsset">
41             <xsd:annotation>
42             <xsd:documentation>
43             NoAsset models the NULL object,
44             it resembles an Asset in its structure.
45             It states that if an Asset is removed,
46             it still retains it ID and may possess a dangling
47             reference to a related instance of Asset.
48             </xsd:documentation>
49             </xsd:annotation>
50             <xsd:sequence>
51                     <xsd:element name="id"
52                                  type="generic:ID"
53                                  minOccurs="1" maxOccurs="1"/>
54                     <xsd:element name="type"
55                                  type="generic:AssetClass"
56                                  minOccurs="0" maxOccurs="1"/>
57             </xsd:sequence>
58     </xsd:complexType>
59     <xsd:simpleType name="ComparisonOperator">
60             <xsd:annotation>
61             <xsd:documentation>this type models the enumeration
62                  type ConstraintDescription.ComparisonOperator
63                  in the generic package</xsd:documentation>
64             </xsd:annotation>
65             <xsd:restriction base="xsd:string">
66                     <xsd:enumeration value="EQUAL"/>
67                     <xsd:enumeration value="GREATER"/>
68                     <xsd:enumeration value="GREATER_OR_EQUAL"/>
69                     <xsd:enumeration value="LESS"/>
70                     <xsd:enumeration value="LESS_OR_EQUAL"/>
71                     <xsd:enumeration value="NOT_EQUAL"/>
72                     <xsd:enumeration value="SIMILIAR"/>
73             </xsd:restriction>
74     </xsd:simpleType>
75     <xsd:complexType name="AbstractAsset" abstract="true">
76             <xsd:sequence>
77                     <xsd:element name="id"
78                                  type="generic:ID"
79                                  minOccurs="1" maxOccurs="1"/>
80                     <xsd:element name="type"
81                                  type="generic:AssetClass"
82                                  minOccurs="0" maxOccurs="1"/>
83             </xsd:sequence>
84     </xsd:complexType>
85     <xsd:complexType name="Asset">
86             <xsd:complexContent>
87                     <xsd:extension base="generic:AbstractAsset"/>
88             </xsd:complexContent>
89     </xsd:complexType>
90     <xsd:complexType name="AssetClass">
```

```
91                <xsd:complexContent>
92                        <xsd:extension base="generic:Asset">
93                                <xsd:sequence>
94                                        <xsd:element name="name"
95                                             type="xsd:string"
96                                             minOccurs="1" maxOccurs="1"/>
97                                        <xsd:element name="superclass"
98                                             type="generic:AssetClass"
99                                             minOccurs="0" maxOccurs="1"/>
100                               </xsd:sequence>
101                       </xsd:extension>
102               </xsd:complexContent>
103       </xsd:complexType>
104       <xsd:complexType name="AssetList">
105               <xsd:sequence minOccurs="0" maxOccurs="unbounded">
106                       <xsd:element name="AssetListElement"
107                                    type="generic:Asset"/>
108               </xsd:sequence>
109       </xsd:complexType>
110       <xsd:complexType name="AssetIterator">
111               <xsd:complexContent>
112                       <xsd:extension base="generic:AssetList"/>
113               </xsd:complexContent>
114       </xsd:complexType>
115       <xsd:complexType name="QueryConstraint">
116               <xsd:sequence>
117                       <xsd:element name="attributeName"
118                                    type="xsd:string"
119                                    minOccurs="0" maxOccurs="1"/>
120                       <xsd:element name="comparator"
121                                    type="xsd:anyType"
122                                    minOccurs="0" maxOccurs="1"/>
123               </xsd:sequence>
124       </xsd:complexType>
125       <xsd:complexType name="CharacteristicQueryConstraint">
126               <xsd:complexContent>
127                       <xsd:extension base="generic:QueryConstraint">
128                               <xsd:sequence>
129                                       <xsd:element name="constrainingValue"
130                                            type="xsd:anyType"
131                                            minOccurs="1" maxOccurs="1"/>
132                               </xsd:sequence>
133                       </xsd:extension>
134               </xsd:complexContent>
135       </xsd:complexType>
136       <xsd:complexType name="ToManyRelationshipQueryConstraint">
137               <xsd:complexContent>
138                       <xsd:extension base="generic:QueryConstraint">
139                               <xsd:sequence>
140                                       <xsd:element name="relatedAssets"
141                                            type="generic:AssetList"
142                                            minOccurs="0" maxOccurs="1"/>
143                               </xsd:sequence>
144                       </xsd:extension>
```

```
145                 </xsd:complexContent>
146         </xsd:complexType>
147         <xsd:complexType name="ToOneRelationshipQueryConstraint">
148                 <xsd:complexContent>
149                         <xsd:extension base="generic:QueryConstraint">
150                                 <xsd:sequence>
151                                         <xsd:element name="relatedAsset"
152                                              type="generic:Asset"
153                                              minOccurs="0" maxOccurs="1"/>
154                                 </xsd:sequence>
155                         </xsd:extension>
156                 </xsd:complexContent>
157         </xsd:complexType>
158         <xsd:complexType name="MemberInitialization">
159                 <xsd:sequence>
160                         <xsd:element name="membername" type="xsd:string"/>
161                 </xsd:sequence>
162         </xsd:complexType>
163         <xsd:complexType name="AttributeInitialization">
164                 <xsd:complexContent>
165                         <xsd:extension base="generic:MemberInitialization"/>
166                 </xsd:complexContent>
167         </xsd:complexType>
168         <xsd:complexType name="CharacteristicInitialization">
169                 <xsd:complexContent>
170                         <xsd:extension base="generic:AttributeInitialization">
171                                 <xsd:sequence>
172                                         <xsd:element name="value"
173                                                      type="xsd:anyType"/>
174                                 </xsd:sequence>
175                         </xsd:extension>
176                 </xsd:complexContent>
177         </xsd:complexType>
178         <xsd:complexType name="ContentInitialization">
179                 <xsd:complexContent>
180                         <xsd:extension base="generic:MemberInitialization">
181                                 <xsd:sequence>
182                                         <xsd:element name="value"
183                                                      type="xsd:anyType"/>
184                                 </xsd:sequence>
185                         </xsd:extension>
186                 </xsd:complexContent>
187         </xsd:complexType>
188         <xsd:complexType name="RelationshipInitialization">
189                 <xsd:complexContent>
190                         <xsd:extension base="generic:AttributeInitialization">
191                                 <xsd:sequence>
192                                         <xsd:element name="value"
193                                                      type="generic:Asset"/>
194                                 </xsd:sequence>
195                         </xsd:extension>
196                 </xsd:complexContent>
197         </xsd:complexType>
198 </xsd:schema>
```

199

Appendix B

XML Schema Generator Configurations

without substitutionGroup

```
1
2  <?xml version="1.0"?>
3  <cat xmlns:util="http://www.sts.tu-harburg.de/2004/java/util/xmlconfigfile">
4   <scanner class="de.tuhh.sts.cocoma.compiler.ADLScanner"/>
5   <parser class="de.tuhh.sts.cocoma.compiler.ADLParser"/>
6   <configuration name="xmlschema">
7    <param name="outputDirBase">gen</param>
8    <generator name="schemagen"
9     class="de.tuhh.sts.cocoma.compiler.generators.xmlschema.ALDToXMLSchemaGenerator">
10    <param name="outputFile">
11     <util:xpath
12      path="../../../param[@name='outputDirBase']/text()"/>/schema/schema.xsd
13    </param>
14    <param name="targetNamespace">urn:de.tuhh.sts.cocoma.generic</param>
15    <param name="prefix">gkns</param>
16    <param name="ignore-constraints">true</param>
17   </generator>
18  </configuration>
19 </cat>
20
```

with substitutionGroup

```
1
2  <?xml version="1.0"?>
3  <cat xmlns:util="http://www.sts.tu-harburg.de/2004/java/util/xmlconfigfile">
4   <scanner class="de.tuhh.sts.cocoma.compiler.ADLScanner"/>
5   <parser class="de.tuhh.sts.cocoma.compiler.ADLParser"/>
6   <configuration name="xmlschema">
7    <param name="outputDirBase">gen</param>
8    <generator name="schemagen"
9     class="de.tuhh.sts.cocoma.compiler.generators.xmlschema.ALDToXMLSchemaGenerator">
10    <param name="outputFile">
11     <util:xpath
12      path="../../../param[@name='outputDirBase']/text()"/>/schema/schema.xsd
13    </param>
14    <param name="targetNamespace">urn:de.tuhh.sts.cocoma.generic</param>
15    <param name="prefix">gkns</param>
```

```
16      <param name="ignore-constraints">true</param>
17      <param name="substitutionGroup">true</param>
18      <param name="typeSuffix"></param>
19     </generator>
20    </configuration>
21   </cat>
22
```

Appendix C

WSDL Generator Configuration

```
1  <cat xmlns:util="http://www.sts.tu-harburg.de/2004/java/util/xmlconfigfile">
2   <scanner class="de.tuhh.sts.cocoma.compiler.ADLScanner"/>
3   <parser class="de.tuhh.sts.cocoma.compiler.ADLParser"/>
4   <configuration name="wsdlgenerator">
5     <param name="outputDirBase">generated</param>
6     <generator name="schemagen"
7      class="de.tuhh.sts.cocoma.compiler.generators.xmlschema.ALDToXMLSchemaGenerator">
8       <param name="outputFile">
9         <util:xpath
10         path="../../../param[@name='outputDirBase']/text()"/>/schema/schema.xsd
11      </param>
12      <param name="targetNamespace">http://www.sts.tu-harburg.de/test/</param>
13      <param name="prefix">tns</param>
14      <param name="ignore-constraints">true</param>
15      <param name="substitutionGroup">true</param>
16      <param name="typeSuffix"></param>
17    </generator>
18    <generator name="ald2wsdlgenerator"
19     class="de.tuhh.sts.cocoma.compiler.generators.ws.wsdl.ALD2WSDLGenerator">
20      <param name="outputFile">
21        <util:xpath path="../../../param[@name='outputDirBase']/text()"/>
22         /schema/wsdl/cocoma_s_module_interface.wsdl
23      </param>
24      <param name="generatorClass">ALD2WSDLGenerator</param>
25      <param name="generatorName">cocoma wsdl generator</param>
26      <param name="generatorType">service module generator</param>
27      <param name="generatorID">wsdlgen_r1</param>
28      <param name="generatorVersion">0.1</param>
29      <param name="generatorAuthor">Patrick Un</param>
30      <param name="generatorConfigurationName">wsdlgenerator</param>
31      <param name="generatorConfigurationPath">config</param>
32      <param name="generatorConfigurationFile">cat_wsdlgen.xml</param>
33      <param name="generatorMappingRulesFile">
34                  config/wsdlgenerator_mapping_rules.xml
35      </param>
36      <param name="generatorMetaMappingRulesFile">
37                  config/wsdlgenerator_mapping_meta_rules.xml
```

```
38        </param>
39        <param name="generatorDependency">schemagen</param>
40        <param name="generatorImportedXSDSchemaFile">schema.xsd</param>
41        <param name="generatorTargetNamespace">urn:de.tuhh.sts.cocoma.generic</param>
42        <param name="generatorTargetNamespacePrefix">tns</param>
43        <param name="generatorWSDLSOAPStyle">document</param>
44        <param name="generatorWSDLversion">1.1</param>
45        <param name="generatorOutput">web services description language</param>
46        <param name="generatorOutputFileType">WSDL</param>
47        <param name="generatorOutputFileName">cocoma_s_module_interface.wsdl</param>
48        <param name="generatorWSDLServiceName">CocomaGenericSModuleService</param>
49        <param name="generatorWSDLServiceURL">
50                    http://localhost:8080/axis/services/CocomaGenericSModuleService
51        </param>
52        <param name="generatorLoggingEnable">true</param>
53        <param name="generatorLoggingLevel">FATAL</param>
54      </generator>
55    </configuration>
56  </cat>
57
```

Appendix D

WSDL Document of Server Module

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="urn:de.tuhh.sts.cocoma.generic"
                  xmlns="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:tns="urn:de.tuhh.sts.cocoma.generic"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  xmlns:generic="urn:de.tuhh.sts.cocoma.generic"
                  xmlns:cocoma="urn:de.tuhh.sts.cocoma.generic"
                  xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
                  xmlns:impl="urn:de.tuhh.sts.cocoma.generic">
        <wsdl:types>
                <xsd:import namespace="urn:de.tuhh.sts.cocoma.generic"
                            schemaLocation="generic_interface_hierarchy.xsd"/>
                <xsd:import namespace="urn:de.tuhh.sts.cocoma.generic.smodule"
                            schemaLocation="schema.xsd"/>
        </wsdl:types>
        <wsdl:message name="getClassRequest">
                <wsdl:part name="argument0" type="xsd:string"/>
        </wsdl:message>
        <wsdl:message name="getClassResponse">
                <wsdl:part name="getClassReturn" type="impl:AssetClass"/>
        </wsdl:message>
        <wsdl:message name="createRequest_1">
                <wsdl:part name="argument0" type="impl:AssetClass"/>
                <wsdl:part name="argument1" type="impl:ArrayOfModuleMemberInitialization"/>
        </wsdl:message>
        <wsdl:message name="createResponse_1">
                <wsdl:part name="createReturn" type="impl:Asset"/>
        </wsdl:message>
        <wsdl:message name="createRequest_2">
                <wsdl:part name="argument0" type="impl:AssetClass"/>
                <wsdl:part name="argument1" type="impl:AbstractAsset"/>
        </wsdl:message>
        <wsdl:message name="createResponse_2">
                <wsdl:part name="createReturn" type="impl:Asset"/>
        </wsdl:message>
```

```
<wsdl:message name="createRequest_3">
        <wsdl:part name="argument0" type="impl:AssetClass"/>
        <wsdl:part name="argument1" type="impl:AssetIterator"/>
</wsdl:message>
<wsdl:message name="createResponse_3">
        <wsdl:part name="createReturn" type="impl:Asset"/>
</wsdl:message>
<wsdl:message name="deleteRequest_1">
        <wsdl:part name="argument0" type="impl:Asset"/>
</wsdl:message>
<wsdl:message name="deleteResponse_1">
        <wsdl:part name="deleteReturn" type="impl:NewAsset"/>
</wsdl:message>
<wsdl:message name="deleteRequest_2">
        <wsdl:part name="argument0" type="impl:AssetIterator"/>
</wsdl:message>
<wsdl:message name="deleteResponse_2">
        <wsdl:part name="deleteReturn" type="impl:AssetIterator"/>
</wsdl:message>
<wsdl:message name="lookforRequest_1">
        <wsdl:part name="argument0" type="impl:ID"/>
</wsdl:message>
<wsdl:message name="lookforResponse_1">
        <wsdl:part name="lookforReturn" type="impl:Asset"/>
</wsdl:message>
<wsdl:message name="lookforRequest_2">
        <wsdl:part name="argument0" type="xsd:string"/>
</wsdl:message>
<wsdl:message name="lookforResponse_2">
        <wsdl:part name="lookforReturn" type="impl:AssetIterator"/>
</wsdl:message>
<wsdl:message name="lookforRequest_3">
        <wsdl:part name="argument0" type="impl:AssetClass"/>
        <wsdl:part name="argument1" type="xsd:string"/>
</wsdl:message>
<wsdl:message name="lookforResponse_3">
        <wsdl:part name="lookforReturn" type="impl:AssetIterator"/>
</wsdl:message>
<wsdl:message name="lookforRequest_4">
        <wsdl:part name="argument0" type="impl:AssetClass"/>
        <wsdl:part name="argument1" type="impl:ArrayOfModuleQueryConstraint"/>
</wsdl:message>
<wsdl:message name="lookforResponse_4">
        <wsdl:part name="lookforReturn" type="impl:AssetIterator"/>
</wsdl:message>
<wsdl:message name="lookforRequest_5">
        <wsdl:part name="argument0" type="impl:AssetClass"/>
        <wsdl:part name="argument1" type="impl:AbstractAsset"/>
</wsdl:message>
<wsdl:message name="lookforResponse_5">
        <wsdl:part name="lookforReturn" type="impl:AssetIterator"/>
</wsdl:message>
<wsdl:message name="lookforRequest_6">
        <wsdl:part name="argument0" type="impl:AssetClass"/>
```

```
        <wsdl:part name="argument1" type="impl:AssetIterator"/>
</wsdl:message>
<wsdl:message name="lookforResponse_6">
        <wsdl:part name="lookforReturn" type="impl:AssetIterator"/>
</wsdl:message>
<wsdl:message name="modifyRequest_1">
        <wsdl:part name="argument0" type="impl:Asset"/>
        <wsdl:part name="argument1" type="impl:ArrayOfModuleMemberInitialization"/>
</wsdl:message>
<wsdl:message name="modifyResponse_1">
        <wsdl:part name="modifyReturn" type="impl:Asset"/>
</wsdl:message>
<wsdl:message name="modifyRequest_2">
        <wsdl:part name="argument0" type="impl:Asset"/>
        <wsdl:part name="argument1" type="impl:AbstractAsset"/>
</wsdl:message>
<wsdl:message name="modifyResponse_2">
        <wsdl:part name="modifyReturn" type="impl:Asset"/>
</wsdl:message>
<wsdl:message name="modifyRequest_3">
        <wsdl:part name="argument0" type="impl:AssetIterator"/>
        <wsdl:part name="argument1" type="impl:ArrayOfModuleMemberInitialization"/>
</wsdl:message>
<wsdl:message name="modifyResponse_3">
        <wsdl:part name="modifyReturn" type="impl:AssetIterator"/>
</wsdl:message>
<wsdl:message name="modifyRequest_4">
        <wsdl:part name="argument0" type="impl:AssetIterator"/>
        <wsdl:part name="argument1" type="impl:AbstractAsset"/>
</wsdl:message>
<wsdl:message name="modifyResponse_4">
        <wsdl:part name="modifyReturn" type="impl:AssetIterator"/>
</wsdl:message>
<wsdl:portType name="CocomaGenericSModulePortType">
        <wsdl:operation name="getClass" parameterOrder="argument0">
                <wsdl:input name="getClassRequest" message="impl:getClassRequest"/>
                <wsdl:output name="getClassResponse" message="impl:getClassResponse"/>
        </wsdl:operation>
        <wsdl:operation name="create" parameterOrder="argument0 argument1">
                <wsdl:input name="createRequest_1" message="impl:createRequest_1"/>
                <wsdl:output name="createResponse_1" message="impl:createResponse_1"/>
        </wsdl:operation>
        <wsdl:operation name="create" parameterOrder="argument0 argument1">
                <wsdl:input name="createRequest_2" message="impl:createRequest_2"/>
                <wsdl:output name="createResponse_2" message="impl:createResponse_2"/>
        </wsdl:operation>
        <wsdl:operation name="create" parameterOrder="argument0 argument1">
                <wsdl:input name="createRequest_3" message="impl:createRequest_3"/>
                <wsdl:output name="createResponse_3" message="impl:createResponse_3"/>
        </wsdl:operation>
        <wsdl:operation name="delete" parameterOrder="argument0">
                <wsdl:input name="deleteRequest_1" message="impl:deleteRequest_1"/>
                <wsdl:output name="deleteResponse_1" message="impl:deleteResponse_1"/>
        </wsdl:operation>
```

```
        <wsdl:operation name="delete" parameterOrder="argument0">
                <wsdl:input name="deleteRequest_2" message="impl:deleteRequest_2"/>
                <wsdl:output name="deleteResponse_2" message="impl:deleteResponse_2"/>
        </wsdl:operation>
        <wsdl:operation name="lookfor" parameterOrder="argument0">
                <wsdl:input name="lookforRequest_1" message="impl:lookforRequest_1"/>
                <wsdl:output name="lookforResponse_1" message="impl:lookforResponse_1"/>
        </wsdl:operation>
        <wsdl:operation name="lookfor" parameterOrder="argument0">
                <wsdl:input name="lookforRequest_2" message="impl:lookforRequest_2"/>
                <wsdl:output name="lookforResponse_2" message="impl:lookforResponse_2"/>
        </wsdl:operation>
        <wsdl:operation name="lookfor" parameterOrder="argument0 argument1">
                <wsdl:input name="lookforRequest_3" message="impl:lookforRequest_3"/>
                <wsdl:output name="lookforResponse_3" message="impl:lookforResponse_3"/>
        </wsdl:operation>
        <wsdl:operation name="lookfor" parameterOrder="argument0 argument1">
                <wsdl:input name="lookforRequest_4" message="impl:lookforRequest_4"/>
                <wsdl:output name="lookforResponse_4" message="impl:lookforResponse_4"/>
        </wsdl:operation>
        <wsdl:operation name="lookfor" parameterOrder="argument0 argument1">
                <wsdl:input name="lookforRequest_5" message="impl:lookforRequest_5"/>
                <wsdl:output name="lookforResponse_5" message="impl:lookforResponse_5"/>
        </wsdl:operation>
        <wsdl:operation name="lookfor" parameterOrder="argument0 argument1">
                <wsdl:input name="lookforRequest_6" message="impl:lookforRequest_6"/>
                <wsdl:output name="lookforResponse_6" message="impl:lookforResponse_6"/>
        </wsdl:operation>
        <wsdl:operation name="modify" parameterOrder="argument0 argument1">
                <wsdl:input name="modifyRequest_1" message="impl:modifyRequest_1"/>
                <wsdl:output name="modifyResponse_1" message="impl:modifyResponse_1"/>
        </wsdl:operation>
        <wsdl:operation name="modify" parameterOrder="argument0 argument1">
                <wsdl:input name="modifyRequest_2" message="impl:modifyRequest_2"/>
                <wsdl:output name="modifyResponse_2" message="impl:modifyResponse_2"/>
        </wsdl:operation>
        <wsdl:operation name="modify" parameterOrder="argument0 argument1">
                <wsdl:input name="modifyRequest_3" message="impl:modifyRequest_3"/>
                <wsdl:output name="modifyResponse_3" message="impl:modifyResponse_3"/>
        </wsdl:operation>
        <wsdl:operation name="modify" parameterOrder="argument0 argument1">
                <wsdl:input name="modifyRequest_4" message="impl:modifyRequest_4"/>
                <wsdl:output name="modifyResponse_4" message="impl:modifyResponse_4"/>
        </wsdl:operation>
</wsdl:portType>
<wsdl:binding name="CocomaGenericSModuleBinding" type="tns:CocomaGenericSModulePortType">
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
        <wsdl:operation name="getClass">
                <soap:operation soapAction="default" style="document"/>
                <wsdl:input name="getClassRequest">
                        <soap:body use="literal"/>
                </wsdl:input>
                <wsdl:output name="getClassResponse">
                        <soap:body use="literal"/>
```

```
                </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="create">
                <soap:operation soapAction="default" style="document"/>
                <wsdl:input name="createRequest_1">
                        <soap:body use="literal"/>
                </wsdl:input>
                <wsdl:output name="createResponse_1">
                        <soap:body use="literal"/>
                </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="create">
                <soap:operation soapAction="default" style="document"/>
                <wsdl:input name="createRequest_2">
                        <soap:body use="literal"/>
                </wsdl:input>
                <wsdl:output name="createResponse_2">
                        <soap:body use="literal"/>
                </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="create">
                <soap:operation soapAction="default" style="document"/>
                <wsdl:input name="createRequest_3">
                        <soap:body use="literal"/>
                </wsdl:input>
                <wsdl:output name="createResponse_3">
                        <soap:body use="literal"/>
                </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="delete">
                <soap:operation soapAction="default" style="document"/>
                <wsdl:input name="deleteRequest_1">
                        <soap:body use="literal"/>
                </wsdl:input>
                <wsdl:output name="deleteResponse_1">
                        <soap:body use="literal"/>
                </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="delete">
                <soap:operation soapAction="default" style="document"/>
                <wsdl:input name="deleteRequest_2">
                        <soap:body use="literal"/>
                </wsdl:input>
                <wsdl:output name="deleteResponse_2">
                        <soap:body use="literal"/>
                </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="lookfor">
                <soap:operation soapAction="default" style="document"/>
                <wsdl:input name="lookforRequest_1">
                        <soap:body use="literal"/>
                </wsdl:input>
                <wsdl:output name="lookforResponse_1">
                        <soap:body use="literal"/>
```

```
        </wsdl:output>
</wsdl:operation>
<wsdl:operation name="lookfor">
        <soap:operation soapAction="default" style="document"/>
        <wsdl:input name="lookforRequest_2">
                <soap:body use="literal"/>
        </wsdl:input>
        <wsdl:output name="lookforResponse_2">
                <soap:body use="literal"/>
        </wsdl:output>
</wsdl:operation>
<wsdl:operation name="lookfor">
        <soap:operation soapAction="default" style="document"/>
        <wsdl:input name="lookforRequest_3">
                <soap:body use="literal"/>
        </wsdl:input>
        <wsdl:output name="lookforResponse_3">
                <soap:body use="literal"/>
        </wsdl:output>
</wsdl:operation>
<wsdl:operation name="lookfor">
        <soap:operation soapAction="default" style="document"/>
        <wsdl:input name="lookforRequest_4">
                <soap:body use="literal"/>
        </wsdl:input>
        <wsdl:output name="lookforResponse_4">
                <soap:body use="literal"/>
        </wsdl:output>
</wsdl:operation>
<wsdl:operation name="lookfor">
        <soap:operation soapAction="default" style="document"/>
        <wsdl:input name="lookforRequest_5">
                <soap:body use="literal"/>
        </wsdl:input>
        <wsdl:output name="lookforResponse_5">
                <soap:body use="literal"/>
        </wsdl:output>
</wsdl:operation>
<wsdl:operation name="lookfor">
        <soap:operation soapAction="default" style="document"/>
        <wsdl:input name="lookforRequest_6">
                <soap:body use="literal"/>
        </wsdl:input>
        <wsdl:output name="lookforResponse_6">
                <soap:body use="literal"/>
        </wsdl:output>
</wsdl:operation>
<wsdl:operation name="modify">
        <soap:operation soapAction="default" style="document"/>
        <wsdl:input name="modifyRequest_1">
                <soap:body use="literal"/>
        </wsdl:input>
        <wsdl:output name="modifyResponse_1">
                <soap:body use="literal"/>
```

```
                </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="modify">
                <soap:operation soapAction="default" style="document"/>
                <wsdl:input name="modifyRequest_2">
                        <soap:body use="literal"/>
                </wsdl:input>
                <wsdl:output name="modifyResponse_2">
                        <soap:body use="literal"/>
                </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="modify">
                <soap:operation soapAction="default" style="document"/>
                <wsdl:input name="modifyRequest_3">
                        <soap:body use="literal"/>
                </wsdl:input>
                <wsdl:output name="modifyResponse_3">
                        <soap:body use="literal"/>
                </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="modify">
                <soap:operation soapAction="default" style="document"/>
                <wsdl:input name="modifyRequest_4">
                        <soap:body use="literal"/>
                </wsdl:input>
                <wsdl:output name="modifyResponse_4">
                        <soap:body use="literal"/>
                </wsdl:output>
        </wsdl:operation>
</wsdl:binding>
<wsdl:service name="CocomaGenericSModuleService">
 <wsdl:port name="CocomaGenericSModuleServicePort"
           binding="tns:CocomaGenericSModuleBinding">
  <soap:address
   location="http://ws.sts.tu-harburg.de:8080/axis/services/CocomaGenericSModuleService"/>
 </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

Appendix E

Code Excerpt

```
public class ProxyAgent {

        private static ProxyAgent instance = null;

        private static HashMap<String,ServerModule> registry = null;

        private static HashMap<String,Component> component = null;

        /**
         * private default constructor
         */
        private ProxyAgent() {
                if (registry == null) {
                        registry = new HashMap<String, ServerModule>();
                }
                if (component == null) {
                        component = new HashMap<String, Component>();
                }
        }

        /**
         *
         * @return singleton instance
         */
        public static ProxyAgent getInstance() {
                if (instance == null) {
                        instance = new ProxyAgent();
                }
                return instance;
        }

        /**
         * @return the registry
         */
        private static HashMap<String, ServerModule> getRegistry() {
                return registry;
        }
```

```
/**
 * @return the component
 */
private static HashMap<String, Component> getComponent() {
        return component;
}

/**
 *
 * @param moduleName
 * @param module
 */
synchronized public void register(String moduleName, ServerModule module) {
        if(moduleName == null || moduleName.length() == 0) {
                return;
        }
        if(module == null || !(module instanceof ServerModule)) {
                return;
        }
        getRegistry().put(moduleName, module);
}

/**
 *
 * @param moduleName
 * @return
 */
public ServerModule lookup(String moduleName) {
        if(moduleName == null || moduleName.length() == 0) {
                return null;
        }
        ServerModule module = (ServerModule)getRegistry().get(moduleName);
        return module;
}

/**
 *
 * @param componentName
 * @param cocomaComponent
 */
synchronized public void registerComponent(String componentName,
                                           Component cocomaComponent) {
        if(componentName == null || componentName.length() == 0) {
                return;
        }
        if(cocomaComponent == null || !(cocomaComponent instanceof Component)){
                return;
        }
        getComponent().put(componentName,cocomaComponent);
}

/**
 *
 * @param componentName
```

```
     * @return
     */
    public Component lookupComponent(String componentName) {
            if(componentName == null || componentName.length() == 0) {
                    return null;
            }
            Component cocomaComponent = (Component)getComponent().get(componentName);
            return cocomaComponent;
    }

}
```

Appendix F

Class Diagrams

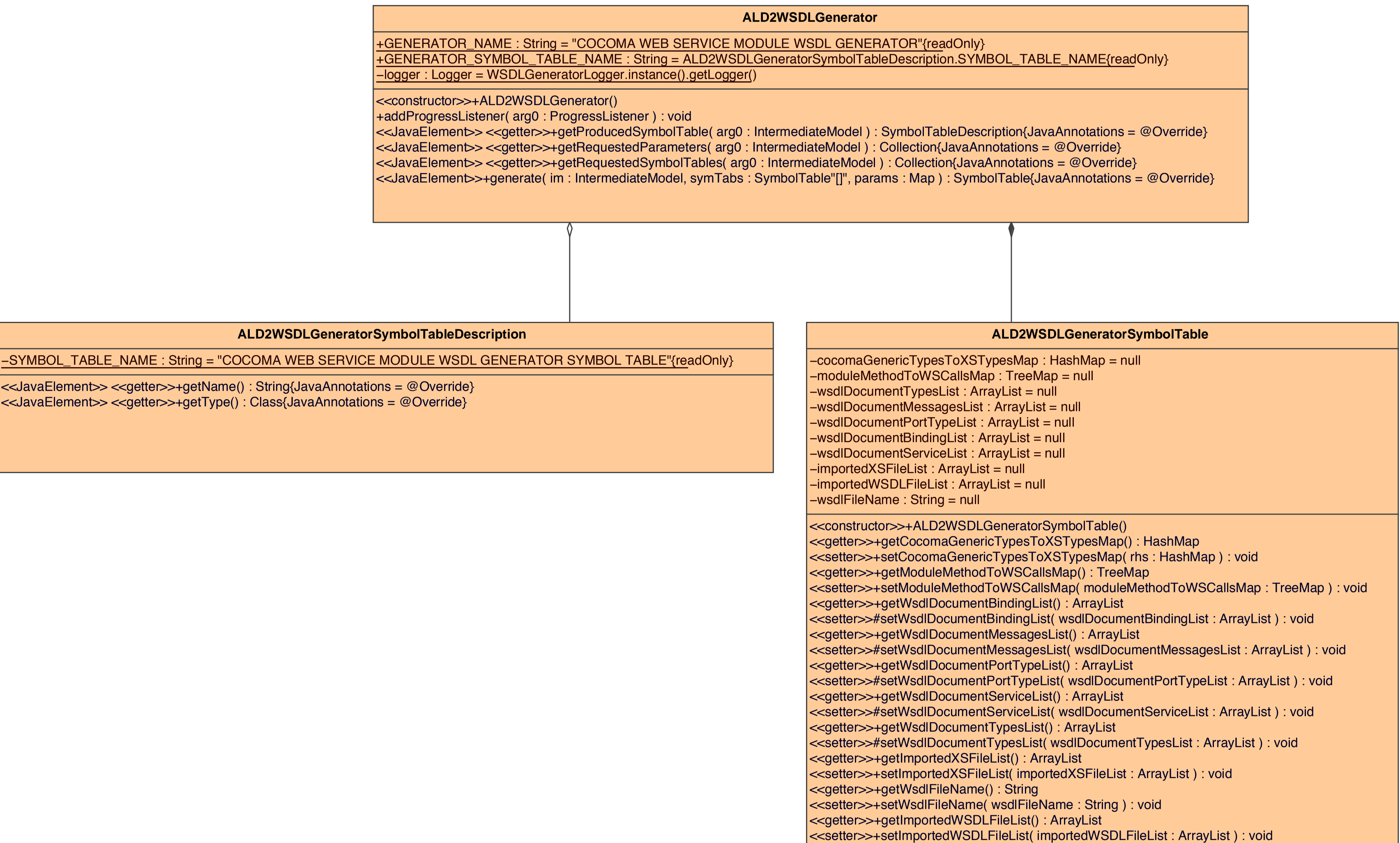


Figure F.1: ALD2WSDLGenerator and ALD2WSDLGeneratorSymbolTable

Figure F.2: WSDL generator classes Part1

Figure F.3: WSDL generator classes Part2

Bibliography

[AM02] Naresh Apte and Toral Mehta. *UDDI: building registry-based web services solutions*. Prentice Hall PTR Inc., One Lake Street Upper Saddle River, NJ 07458, USA, 1st edition, December 2002.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented software architecture: a system of patterns*. Wiley Publishing Inc., 10475 Crosspoint Blvd. Indianapolis, IN 46256, USA, 1st edition, Feb 1996.

[BN97] Philip Bernstein and Eric Newcomer. *Principles of transaction processing*. Morgan Kaufmann Publishers, an imprint of Elsevier Science, 340 Pine Street, 6th floor, San Francisco, CA 94104-3205, USA, 1st edition, Feb 1997.

[Bos03] Sebastian Bossung. Semi-automatic discovery of mapping rules to match xml schemas. Studienarbeit, The University of Auckland, November 2003.

[Bos04] Sebastian Bossung. Generating schema information for views over semistructured data. Diplomarbeit, TU Hamburg-Harburg, July 2004.

[BSHS06] Sebastian Bossung, Hans-Werner Sehring, Patrick Hupe, and Joachim W. Schmidt. Open and dynamic schema evolution in content-intensive web applications. In Jose Cordeiro, Vitor Pedrosa, Bruno Encarnacao, and Joaquim Filipe, editors, *Proceedings of the Second International Conference on Web Information Systems and Technologies*, pages 109–116. INSTICC, INSTICC Press, April 2006.

[Cas55] Ernst Cassirer. *Philosophy of Symbolic Forms*. Yale University Press, Yale USA, 1st edition, 1955.

[Cas01] Ernst Cassirer. *The language, philosophy of symbolic forms*. Felix Meiner Verlag GmbH, 2001.

[Con00] World Wide Web Consortium. Soap messages with attachments, 2000. http://www.w3.org/TR/SOAP-attachments.

[Con01] World Wide Web Consortium. Web services description language (wsdl) 1.1, March 2001. http://www.w3.org/TR/wsdl.

[Con02] World Wide Web Consortium. Web services activity, 2002. http://www.w3.org/2002/ws/Activity.

[Con03a] World Wide Web Consortium. Soap 1.2 adjuncts, 2003. http://www.w3.org/TR/soap12-part2/.

[Con03b] World Wide Web Consortium. Soap 1.2 messaging framework, 2003. http://www.w3.org/TR/soap12-part1/.

[Con03c] World Wide Web Consortium. Soap 1.2 primer, 2003. http://www.w3.org/TR/soap12-part0/.

[Con04a] World Wide Web Consortium. The xml schema language part0 primer, October 2004. http://www.w3.org/TR/xmlschema-0/.

[Con04b] World Wide Web Consortium. The xml schema language part0 to 2, October 2004. Primer: (http://www.w3.org/TR/xmlschema-0/), Structures: (http://www.w3.org/TR/xmlschema-1/), Datatypes: (http://www.w3.org/TR/xmlschema-2/).

[Con05] World Wide Web Consortium. Web services addressing (ws-addressing), 2005. http://www.w3.org/Submission/ws-addressing/.

[Con06a] World Wide Web Consortium. Web services description language (wsdl) 2.0 part0 primer, March 2006. http://www.w3.org/TR/wsdl20-primer/.

[Con06b] World Wide Web Consortium. Web services description language (wsdl) 2.0 part1 core language, March 2006. http://www.w3.org/TR/wsdl20/.

[Con06c] World Wide Web Consortium. Web services description language (wsdl) 2.0 part2 adjuncts, March 2006. http://www.w3.org/TR/wsdl20-adjuncts/.

[Con06d] World Wide Web Consortium. Web services policy 1.2 framework (ws-policy), 2006. http://www.w3.org/Submission/WS-Policy/.

[dev04] IBM developerWorks. International business machine corporation developerworks – web services security, 2004. http://www-128.ibm.com/developerworks/library/specification/ws-secure/.

[dev05a] IBM developerWorks. International business machine corporation developerworks – business process execution language for web services 1.1, 2005. ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf.

[dev05b] IBM developerWorks. International business machine corporation developerworks – web services atomic transaction specification, 2005. ftp://www6.software.ibm.com/software/developer/library/WS-AtomicTransaction.pdf.

[dev05c] IBM developerWorks. International business machine corporation developerworks – web services business activity specification, 2005. ftp://www6.software.ibm.com/software/developer/library/WS-BusinessActivity.pdf.

[dev05d] IBM developerWorks. International business machine corporation developerworks – web services coordination specification, 2005. ftp://www6.software.ibm.com/software/developer/library/WS-Coordination.pdf.

[dev05e] IBM developerWorks. International business machine corporation developerworks – web services reliable messaging, 2005. http://www-128.ibm.com/developerworks/library/specification/ws-rm/.

[dev05f] IBM developerWorks. International business machine corporation developerworks – web services reliable messaging specification, 2005. ftp://www6.software.ibm.com/software/developer/library/ws-reliablemessaging200502.pdf.

[dev06] IBM developerWorks. International business machine corporation developerworks – web services definition, 2006. http://www-128.ibm.com/developerworks/webservices/newto/websvc.html.

[Erl04] Thomas Erl. *service-oriented architecture: a field guide to integrating xml and web services.* Prentice Hall PTR Inc., Pearson Education Inc. Upper Saddle River, NJ 07458, USA, 1st edition, April 2004.

[Fou06] Apache Software Foundation. Apache axis, 2006. http://ws.apache.org/axis/java/index.html.

[ftAoSISO02] Organization for the Advancement of Structured Information Standards (OASIS). Uddi technical specifications, 2002. http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm.

[ftAoSISO04] Organization for the Advancement of Structured Information Standards (OASIS). Web services security (ws-security), 2004. http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf.

[ftAoSISO06a] Organization for the Advancement of Structured Information Standards (OASIS). Universal description discovery and integration oasis, 2006. http://uddi.org/.

[ftAoSISO06b] Organization for the Advancement of Structured Information Standards (OASIS). Web services trust 1.3 (ws-trust), 2006. http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-trust-1.3-spec-cd-01.pdf.

[ftAoSISOwssw06] Organization for the Advancement of Structured Information Standards (OASIS) web services security (wss). Ws-security core specification 1.1, 2006. http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf.

[GD02] Brian Gibb and Suresh Damodaran. *ebXML: concepts and application.* Wiley Publishing Inc., 10475 Crosspoint Blvd. Indianapolis, IN 46256, USA, 1st edition, October 2002.

[GDS04] Steve Graham, Doug Davis, and Simeon Simeonov. *Building web services with java: making sense of xml, soap, wsdl, and uddi.* Sams Publishing, 201 West 103rd Street, Indianapolis, IN 46290, USA, 2nd edition, June 2004.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: elements of reusable object-oriented software.* Addison Wesley, Pearson Educations Inc., Pearson Education, Inc. One Lake Street Upper Saddle River, NJ 07458, USA, 1st edition, June 1994. Addison Wesley Professional Computer Series.

[Gri98] Frank Griffel. *Componentware: Konzept und Techniken eines Softwareparadigmas.* dpunkt Verlag, Heidelberg, Germany, 1st edition, 1998.

[HWB04] Gregor Hohpe, Bobby Woolf, and Kyle Brown. *Enterprise Integration Patterns: designing, building and deploying messaging solutions.* Addison Wesley, Pearson Educations Inc., Pearson Education, Inc. One Lake Street Upper Saddle River, NJ 07458, USA, 1st edition, June 2004. Addison Wesley Signature Series.

[jcp06] The java community process. Jcp jax-rpc, 2006. https://jax-rpc.dev.java.net/.

[KWW01] Alan Kotok, David Webber, and David R. Webber. *ebXML: the new global standard for doing business on the internet.* Sams Publishing, 201 West 103rd Street, Indianapolis, IN 46290, USA, 1st edition, August 2001.

[Lar98] Craig Larman. *Applying UML and patterns: an introduction to object-oriented analysis and design.* Prentice Hall PTR Inc., Pearson Education Inc. Upper Saddle River, NJ 07458, USA, 2st edition, April 1998.

[Mah04] Qusay H. Mahmoud. Developing web services with java 2 enterprise edition platform, 2004. http://java.sun.com/developer/technicalArticles/J2EE/j2eews/.

[Mic06] Sun Microsystems. java api for xml-based rpc, 2006. http://java.sun.com/webservices/jaxrpc/.

[MTSM03] James McGovern, Sameer Tyagi, Michael Stevens, and Sunil Matthew. *Java web services architecture.* Morgan Kaufmann Publishers, an imprint of Elsevier Science, 340 Pine Street, 6th floor, San Francisco, CA 94104-3205, USA, 1st edition, Jan 2003.

[MW06] Thomas Mattern and Dan Woods. *Enterprise SOA: designing IT for business innovation.* O'Reilly Media Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, USA, 1st edition, 2006.

[Nl04] Eric Newcomer and Greg lomow. *Understanding SOA with web services.* Addison Wesley, Pearson Education, Inc., Pearson Education, Inc. One Lake Street Upper Saddle River, NJ 07458, USA, 1st edition, 2004.

[omg98] OMG The object management group. Corba iiop specification, 1998. http://www.omg.org/technology/documents/formal/.

[SBMS04] Inderjeet Singh, Sean Brydon, Greg Murray, and Beth Stearns. *Designing web services with the J2EE 1.4 Platform.* Addison Wesley, Pearson Educations Inc., Pearson Education, Inc. One Lake Street Upper Saddle River, NJ 07458, USA, 1st edition, June 2004.

[Seh04] Hans-Werner Sehring. *Konzeptorientierte Inhaltsverwaltung Modell Systemarchitektur und Prototypen Hamburg University of Technology.* PhD thesis, University of Technology Hamburg, STS Institute for Software Systems, Harburger Schlossstr. 20, 21073 Hamburg, February 2004. dissertation.de - Verlag im Internet GmbH.

[SG96] Mary Shaw and David Garlan. *Software architecture, perspective on an emerging discipline.* Prentice-Hall Inc., Upper Saddle River, New Jersey, USA, 1st edition, 1996.

[SM06] Inc. Sun Microsystems. Java rmi tutorial, 1995-2006. http://java.sun.com/docs/books/tutorial/rmi/index.html.

[SS03] Joachim W. Schmidt and Hans-Werner Sehring. Conceptual content modeling and management the rationale of an asset language. In Manfred Broy and Alexandre V. Zamulin, editors, *volume 2890 of Lecture Notes in Computer Science*, page 469. Springer Verlag, 2003. Proceedings Perspectives of Systems Informatics Novosibirsk, Akademgorodok, Russia, July 9-12.

[Ton06] Kent Tong. *Developing web services with Apache AXIS.* LuLu Press, P.O. Box 2344, Napa, CA 94558, USA, 1st edition, April 2006.

[Wal02] Aaron E. Walsh. *ebXML: the technical specifications.* Prentice Hall PTR Inc., One Lake Street Upper Saddle River, NJ 07458, USA, 1st edition, January 2002.