Implementation of Information Dissemination System using Aspect-Oriented Programming

by

Flannan Lo

A thesis submitted to fulfill the equivalent of three half courses of the requirements for the degree of Master of Engineering in the

Edward S. Rogers Sr. Department of Electrical and Computer Engineering

University of Toronto

© Copyright by Flannan Lo
(2007)
Abstract

Implementation of the Information Dissemination System using Aspect-Oriented Programming

Flannan Lo

Master of Engineering

Edward S. Rogers Sr. Department of Electrical and Computer Engineering

University of Toronto

2007

As a new programming methodology, aspect-oriented programming has been successfully applied to many projects based on object-oriented programming languages such as Java and C++. Since aspect-oriented programming is still very new to procedural languages such as C, there are not many procedural language based projects that are designed and implemented using this programming methodology. In fact, most of these projects are refactoring projects. That is, developers identify and refactor the aspects from an existing system. This research takes an alternative approach to explore the benefits of applying aspect-oriented programming on procedural languages by designing and implementing a large-scale system from scratch. This project experiments with aspect-oriented programming throughout the whole development cycle; that is, from requirements to design to implementation and to testing of a large-scale system, the Information Dissemination (INFOD) system. C and Aspect-oriented C are the main programming languages used in this project.
Table of Contents

1 Introduction ................................................................................................................. 1
2 Background and Related Work ................................................................................... 4
  2.1 Information Dissemination System ...................................................................... 4
  2.2 Aspect Oriented Programming ............................................................................ 7
  2.3 Aspect-oriented C Language ............................................................................... 8
  2.4 Related Work ...................................................................................................... 9
    2.4.1 An Earlier Version of AspectC ................................................................... 9
    2.4.2 A Case Study in Building Application Using AOP .................................... 9
3 Requirements and Design ......................................................................................... 12
  3.1 Core Architecture .............................................................................................. 16
    3.1.1 Registry Manager ...................................................................................... 16
    3.1.2 Publisher Manager .................................................................................... 17
    3.1.3 Consumer Manager ................................................................................... 17
    3.1.4 Subscriber Manager .................................................................................. 17
    3.1.5 Subscription Manager ............................................................................... 17
    3.1.6 Vocabulary Manager ................................................................................. 18
    3.1.7 Association Manager ................................................................................. 18
    3.1.8 Utilities ...................................................................................................... 18
    3.1.9 XML Parser ............................................................................................... 19
  3.2 Message Flow within the INFOD Core System ............................................... 19
    3.2.1 Creating a new Publisher Entity in the INFOD System ........................... 19
    3.2.2 Creating a new Subscription Entity in the INFOD System ...................... 20
    3.2.3 Updating Data Entries in the INFOD System........................................... 22
  3.3 INFOD System Aspects .................................................................................... 23
    3.3.1 Attach SOAP Header Aspect .................................................................... 25
    3.3.2 Remove SOAP Header Aspect .................................................................. 26
    3.3.3 Message Filtering Aspect ......................................................................... 28
    3.3.4 Drop Unused INFOD Elements Aspect .................................................... 29
    3.3.5 Disable New INFOD Elements Aspect ..................................................... 30
    3.3.6 Manipulate Registry Aspect ...................................................................... 32
    3.3.7 Authorization Failure Aspect .................................................................... 33
    3.3.8 Error Handling Aspect .............................................................................. 34
    3.3.9 Logging and Tracing Aspect ...................................................................... 35
    3.3.10 Remove INFOD References Aspect ......................................................... 36
    3.3.11 Remove Namespace Aspect ..................................................................... 37
  3.4 External Services ............................................................................................... 38
    3.4.1 XML Parser ............................................................................................... 39
    3.4.2 XQuery Parser ........................................................................................... 39
    3.4.3 Database .................................................................................................... 40
  3.5 Non-Functional Requirements .......................................................................... 41
    3.5.1 Remote Access to INFOD System ............................................................ 41
    3.5.2 High Performance ..................................................................................... 42
4 Implementation ......................................................................................................... 45
4.1 Attach SOAP Header Aspect ................................................................. 45
4.2 Remove SOAP Header Aspect .............................................................. 47
4.3 Message Filtering Aspect ................................................................. 48
4.4 Drop Unused INFOD Elements Aspect ............................................. 51
4.5 Disable New INFOD Elements Aspect ............................................ 53
4.6 Authorization Failure Aspect .............................................................. 55
4.7 Error Handling Aspect ...................................................................... 57
4.8 Logging and Tracing Aspect .............................................................. 58
4.9 Remove INFOD References Aspect ..................................................... 59
4.10 Remove Namespace Aspect ............................................................... 61
4.11 Client Socket Aspect ....................................................................... 62
4.12 High Performance Aspect ................................................................. 63
5 System Product Family ..................................................................... 66
6 Test Strategy ...................................................................................... 70
6.1 Approach .......................................................................................... 70
6.2 Verification Aspects ........................................................................... 72
6.2.1 Attach SOAP Header Verification Aspect ........................................ 72
6.2.2 Remove SOAP Header Verification Aspect ..................................... 73
6.2.3 Message Filtering Verification Aspect ............................................. 74
6.2.4 Drop Unused INFOD Elements Verification Aspect ....................... 75
6.2.5 Disable New INFOD Elements Verification Aspect ....................... 76
6.2.6 Authorization Failure Verification Aspect ....................................... 77
6.2.7 Error Handling Verification Aspect ................................................ 77
6.2.8 Remove INFOD References Verification Aspect .............................. 78
6.2.9 Remove Namespace Verification Aspect ......................................... 79
7 Evaluation .......................................................................................... 81
7.1 Lessons Learned ............................................................................... 81
7.2 Implementation Evaluation ............................................................... 82
7.3 Performance Evaluation ................................................................. 83
7.4 Evaluation on the Approaches to Identify Aspects ............................. 85
7.5 Missing Requirements in INFOD Base Specification ......................... 86
7.5.1 Manipulating Data Sources ........................................................... 86
7.5.2 Authorization Requirement ............................................................ 87
7.6 Aspect-Oriented Design ................................................................. 87
7.6.1 Logging and Tracing ................................................................. 88
7.6.2 Realizing Crosscutting Concerns ............................................... 88
7.7 Aspect-oriented C Language ............................................................ 89
7.7.1 Aspects of Aspects ................................................................. 89
7.7.2 Accessing Local Variables ............................................................ 92
7.7.3 Modifying Results in After() Advice ........................................... 93
7.8 Idioms ............................................................................................ 94
8 Conclusions ...................................................................................... 96
Bibliography ......................................................................................... 99
Appendix A .......................................................................................... 101
A.1 Use Case Diagram ........................................................................... 101
A.2 Analyzing Use Cases to Identify Aspects ........................................ 114
A.3 Makefile Example ........................................................................................ 130
A.4 Test Data Example ....................................................................................... 133
A.5 README to Configure INFOD System and Run Test Cases ....................... 134
A.6 Libxml2 ......................................................................................................... 135
List of Figures

Figure 1: Overview of INFOD system, adopted from [1]. ................................................. 4
Figure 2: The class diagram describes the core of the INFOD system. ....................... 16
Figure 3: Sequence diagram for creating a new publisher entity. ............................... 19
Figure 4: Sequence diagram for creating a new subscription entity. ............................. 20
Figure 5: Sequence diagram for updating data entries in INFOD system. .................... 22
Figure 6: This diagram illustrates how aspects intercept the message flow between the external services and the INFOD core system. ................................................. 23
Figure 7: This diagram illustrates how the aspects intercept the message flow when dropping an INFOD entity from the INFOD registry. ........................................ 24
Figure 8: This diagram illustrates how the logging and tracing aspect interacts with the INFOD core system. ................................................................. 24
Figure 9: This diagram illustrates how the message filtering aspect interacts with the INFOD core system. ................................................................. 25
Figure 10: The line number at where the Attach SOAP Header aspect is woven into the core. .............................................................................................. 26
Figure 11: The line number at where the Remove SOAP Header aspect is woven into the core. .............................................................................................. 27
Figure 12: The line number at where the Message Filter aspect is woven into the core. .. 29
Figure 13: The line number at where the Drop Unused INFOD Elements aspect woven into the core. ........................................................................... 30
Figure 14: The line number at where the Disable New INFOD Elements aspect is woven into the core. ........................................................................... 31
Figure 15: The line number at where the Authorization Failure aspect is woven into the core. ......................................................................................... 33
Figure 16: The line number at where the Exception Handling aspect is woven into the core. ......................................................................................... 34
Figure 17: The line number at where the Logging and Tracing aspect is woven into the core. ......................................................................................... 36
Figure 18: The line number at where the Remove INFOD References aspect is woven into the core. ........................................................................... 37
Figure 19: The line number at which the Remove Namespace aspect is woven into the core.............................................................................................. 38
Figure 20: Remote access to INFOD system. ................................................................ 41
Figure 21: The line number at which the Client Socket aspect is woven into the core. .... 42
Figure 22: Interprocess communication in INFOD system. .......................................... 43
Figure 23: The line number at which the High Performance aspect is woven into the core. ......................................................................................... 44
Figure 24: Feature diagram of the INFOD system. ......................................................... 66
Figure 25: The use case describing how different actors create an INFOD entity in the INFOD system. ................................................................. 101
Figure 26: The use case describing how different actors replace an INFOD entity in the INFOD system. ................................................................. 102
Figure 27: The use case that describing how different actors remove an INFOD entity in the INFOD system. ................................................................. 103
Figure 28: The use case describing how different actors register a new vocabulary in the INFOD system. ................................................................. 104
Figure 29: The use case describing how different actors remove a vocabulary from the INFOD system. ................................................................. 105
Figure 30: The use case describing how different actors create a property vocabulary instance in the INFOD system. ........................................ 106
Figure 31: The use case describing how different actors remove a property vocabulary instance from the INFOD registry. ............................. 107
Figure 32: The use case describing how different actors create a vocabulary association in the INFOD registry. ........................................................ 108
Figure 33: The use case describing how different actors remove a vocabulary association from the INFOD registry. ........................................ 109
Figure 34: The use case describing how a subscriber creates a subscription in the INFOD registry. ................................................................. 110
Figure 35: The use case describing how a subscriber replaces a subscription in the INFOD registry. ................................................................. 111
Figure 36: The use case describing how a subscriber removes a subscription from the INFOD registry. ................................................................. 112
List of Tables

Table 1: Public APIs in Publisher Manager................................................................. 12
Table 2: Public APIs in Subscriber Manager............................................................... 12
Table 3: Public APIs in Consumer Manager. ............................................................... 13
Table 4: Public APIs in Subscription Manager............................................................ 13
Table 5: Public APIs in Vocabulary Manager. ............................................................. 13
Table 6: Public APIs in Registry Manager. ................................................................. 13
Table 7: This table lists the horizontal decomposition principles, adopted from [4]. ..... 15
Table 8: The table summarizes the aspects identified in the INFOD system. ............ 45
Table 9: The calculation of the number of INFOD system configurations. ............... 67
Table 10: Examples of the INFOD system family members. ...................................... 68
Table 11: The differences between AOP-based and non-AOP based test approaches. .. 71
Table 12: Number of aspects in the INFOD system. ................................................... 82
Table 13: Number of lines of code in the INFOD system. ........................................... 83
Table 14: Time required to build and test the INFOD system................................. 83
Table 15: Test case results of INFOD system. ............................................................ 83
Table 16: Performance metrics on handling 500 requests. ........................................ 84
Table 17: Performance metrics on handling 1000 requests. ...................................... 84
Table 18: Performance metrics on handling 2000 requests. ...................................... 85
1 Introduction

Aspect-oriented programming is a new programming technique that allows separating concerns that cut across a software system. It helps to simplify the code and modularize the software system. There are already some researches done on aspect-oriented programming but most of them are done in AspectJ, an extension of aspect-oriented programming support to Java programming language. In addition, many of these projects are refactoring projects, meaning that they apply the aspect-oriented programming model on an existing system. As aspect-oriented programming becomes more mature on object-oriented programming languages, people becomes more interested on how this new programming technique can impact on software system programmed on procedural programming languages, such as C. This project explores the impact of applying the aspect-oriented programming model on a large-scale system during the entire software development cycle, i.e., from requirement to design, design to implementation, implementation to testing and testing to deployment. This project is implemented in Aspect-oriented C, an extension to C to support aspect-oriented programming.

There are a few challenges need to overcome in this project. The first challenge is how to efficiently identify aspects from the INFOD base specifications. At the early stage of this project, there are only two documents given that describe the behaviour of the system: the Information Dissemination in the Grid Environment – Base Specifications [1] and the INFOD Base Use Case Scenarios [2]. Therefore, we need to draw the requirements from these two documents and then identify the aspects from these requirements. The second
challenge in this project is how to efficiently test the INFOD system. After identifying 12 aspects from the INFOD system, it becomes very difficult to test, in a conventional way, all the system configurations produced by weaving different aspects in the INFOD base system.

This thesis makes a few contributions. By using the two approaches, the use case scenario approach described in [3] and the approach derived from the horizontal decomposition principles described in [4], 12 aspects are successfully identified in the software system. By configuring the system with the 12 identified aspects, it successfully generates a system product family which contains 1024 different configurations. By using the aspect-oriented programming model, a new test strategy is come up to test the different versions of the software system and thus simplifies the testing process by reducing the number of test cases required to test the functionalities offered in the system product family. Furthermore, this thesis presents some additional guidelines on how to implement a large-scale software system using the aspect-oriented programming model. By doing a performance evaluation on the aspect version and the features built-in version of the software system, this thesis illustrates that the overhead introduced by the woven aspects is very small. This thesis also presents some insights and observations found on the aspect-oriented programming and the Aspect-oriented C programming language while implementing the software system.

This thesis is organized as follows. In the next chapter, it provides a brief background on the software system that is implemented in this project, the aspect-oriented programming
and the Aspect-oriented C; this chapter also includes some discussions on related work.

In chapter 3, it describes the requirements and design of the software system. Then in chapter 4, it describes in details on the implementation of the software system. In chapter 5, it describes how each product family member is generated by configuring the software system differently. Chapter 6 discusses the test strategy used to test the overall system.

Chapter 7 is the evaluation chapter which contains some quantitative results measured in the system and also some insights and observations found while working on this project. Finally, the discussion of future work and the conclusion can be found in chapter 8 and chapter 9 respectively.
2 Background and Related Work

2.1 Information Dissemination System

Information Dissemination (INFOD) is a specification of a system that uses a notification mechanism to make data sources available to consumers [1]. The system monitors any state change of data that may generate events. The events may then generate messages, which are created by publishers and sent to consumers. The INFOD system is composed of different INFOD objects: registry\(^1\), publishers\(^2\), consumers\(^3\), subscribers\(^4\), subscriptions\(^5\) and vocabularies\(^6\).

![INFOD System Diagram](image)

**Figure 1: Overview of INFOD system, adopted from [1].**

1. A repository of object designed to match for information providers and information consumers.
2. A publisher is able to create and deliver data in the form of messages to consumers.
3. A consumer is an entity that is able to receive messages delivered by publishers.
4. A subscriber is an entity that can specify subscriptions.
5. A subscription is the primary mean of specifying message flow from publishers to consumers.
6. A vocabulary defines the structure of data. It is used to facilitate a common understanding of data between publishers, consumers and subscribers.
Registry: The registry is the data centre of the INFOD system. It is a resource manager that manages entities, data vocabularies and their associations, property vocabularies and their instances.

Entities: An Entity exists in the registry and contains the information about an external object. Each entity is identified by a unique explicit reference (EPR\textsuperscript{1}) and may also reference the EPR of the corresponding resource. There are three operations in the registry that are used to manage the entities. The create operation allows users to create a new entity for a corresponding external resource in the registry. The replace operation allows users to replace the metadata information contained in an existing entity. The drop operation allows users to remove an existing entity from the registry.

Data vocabularies and their associations: A data vocabulary describes the structure of data that is shared by the INFOD objects. The vocabulary can be specified in different formats: SQL, XML, RDF or other data models. An association allows the user to associate a data vocabulary and an entity in the registry. Associations are generally used to identify a data source as a source of some specific type of information.

Property vocabularies and their instances: A property vocabulary specifies the structure of property constraints associated to entities or vocabulary associations. Unlike the data vocabularies, these vocabularies are always expressed in XML. Property vocabulary instances are instances of property vocabularies registered in the INFOD system. These

\textsuperscript{1} EPR stands for Explicit Reference.
instances represent the values of property that can be used to define extra characteristics of an entity or association.

There are four types of entities that can be found in the INFOD system: Publishers, Consumers, Subscribers and Subscriptions.

Publishers are able to create and deliver data in the form of messages to consumers. Property vocabularies can be used to define extra characteristics of publishers. The messages sent by a publisher can be delivered unconditionally or depending on subscriptions. Property vocabularies can also be used to define constraints to restrict INFOD entities it communicates with.

Consumers are able to receive messages delivered by publishers. Similar to publishers, property vocabularies can be used to define extra characteristics of consumers and they can also be used to define constraints to identify which INFOD entities can interact with them.

Subscribers are the only entities that are allowed to specify subscriptions. A subscription is the primary means of specifying the message flow from publishers to consumers. Property vocabularies can also be used to define constraints to limit which INFOD entities can interact with these subscribers.
Subscriptions only exist in the INFOD registry. They specify what information has to be
delivered for which publishers to which consumers. Unlike other types of entities,
subscriptions do not reference any external object. Instead, they must reference the
subscriber which generates them. Property vocabularies can be used to identify the
publishers and consumers for a subscription. Subscriptions use data constraints to tell
which information is of interest to the referencing subscriber and use dynamic consumer
constraints to specify which consumers to receive a specific message.

2.2 Aspect Oriented Programming

The INFOD system is implemented using aspect-oriented programming.

“Aspect oriented programming offers a new paradigm for software development
by complementing conventional programming paradigms with a higher degree of
separation of concerns. Examples of aspects include security, reliability,
manageability and further non-functional requirements. The existence of aspects
is attributed to the use of the vertical decomposition paradigm to handle
crosscutting concerns in software architectures. AOP overcomes this limitation
by providing new language level constructs to modularize crosscutting concerns”
[4].

There are a number of programming languages available to implement the aspect-
oriented programming model such as AspectJ and Aspect-oriented C.
2.3 Aspect-oriented C Language

The INFOD system is implemented in Aspect-oriented C (ACC).

“Aspect-oriented C is an implementation of aspect-oriented programming (AOP) for the C programming language. Aspect-oriented C is an extension to C. The Aspect-oriented C specification is implemented by the ACC compiler that weaves code written in ACC into ACC-unaware ANSI-C code, and generates C sources implementing the aspect-oriented program. These sources can be compiled by any ANSI-C compliant compiler such as gcc. The current ACC language design adapts the ideas of aspect-oriented programming laid-out in the original paper by Kiczales et al. to the C programming language. The ACC language loosely follows the AspectJ programming language design and the partial ACC language design originally suggested by Coady et al.” [7]

Aspect-oriented C mainly deals with two types of files: aspect files, which implement the aspects, and the core files, which implement the core program. A core program contains a limited number of essential functionalities of a software system that serves as the base of the system to allow further customization [4]. The aspect files contain join points and advices. The aspects can alter the behaviour of the base code by applying advices before, after or in place of the program execution when a join point is reached. Furthermore, Aspect-oriented C can also declare new fields in a C structure that can only be seen within an aspect.
2.4 Related Work

This section summarizes other research projects in aspect-oriented programming that are similar to what have been done in this project.

2.4.1 An Earlier Version of AspectC

There is an earlier version of AspectC proposed by Yvonne Coady and Gregor Kiczales. This version of AspectC was developed as a subset of AspectJ. The main purpose of this AspectC is to help them to determine if aspect-oriented programming can be used to improve the modularity of the operating system code. Specifically, they want to use AspectC to determine whether path-specific customizations can be considered to be crosscutting concerns and whether these concerns can be modularized using aspect-oriented programming [10]. Since most of the operating system code is implemented in C, AspectC becomes a mandatory tool to apply aspect-oriented programming on operating system code. Because it is a subset of AspectJ, the syntax of AspectC is very similar to that of AspectJ. In their project, they were using AspectC to refactor existing code within the page fault handling in the FreeBSD operating system kernel and then hand-compiling the code to C. No AspectC compiler has been release by the group to date.

2.4.2 A Case Study in Building Application Using AOP

There is a case study done by Mik Kersten and Gail Murphy in building a web-based learning environment using aspect-oriented programming. They were building a web-

---

1 AspectJ is an aspect-oriented extension to the Java programming language created at Xerox PARC.
2 FreeBSD is an advanced BSD UNIX® operating system for the Intel® compatible (x86), DEC Alpha and PC-98 architectures.
based application on the Advanced Teaching and Learning Academic Server (ATLAS) to allow students to register courses and navigate through personalized views of course material. By implementing this project, they were investigating whether aspect-oriented programming could impact their design decisions and thus address some of the difficulties being faced in their initial implementation. They were using Xerox PARC’s AspectJ to implement their project. In this case study, they were able to come up with some guidelines that would help them to start building another system using aspect-oriented programming. First, they should try to decouple the aspects and the classes. In other words, the aspects should know nothing about the classes and vice versa. This makes the system easier to evolve in the future. Second, an aspect should act as a factory to simplify the extension of an object’s behaviour. This makes the aspect code simple and clear. Third, developers should avoid using dynamic aspects too often because this complicates the system set-up code. Although dynamic aspects provide more long-term flexibility and support more sophisticated runtime behaviour, it requires weaving in additional registration code, which may not fit the existing system structure easily. Last, aspects should be treated as an extension to the base system model. The base system should be standalone so that when a problem was found after an aspect was woven in, it could easily isolate the problem [11]. Some of the lessons learned in this case study are similar to the lessons learned in this project. Like the first guideline described above, aspects and classes should not know each other. This must be true in implementing this project because all the INFOD system aspects can be taken off from the INFOD system. Otherwise, they need to stick with the core of the INFOD system. Similarly, the last guideline described above must also be true in this project to help debugging easier.
When implementing the INFOD system, the core of the system is implemented first and fully tested before the system aspects are implemented. In this case, if a problem occurs when an aspect is woven in the INFOD system, the problem is mostly caused by the aspect and not the core of the system. For more details about the lessons learned in this project, please refer to Section 7.1 Lessons Learned.
3 Requirements and Design

The INFOD Base Specifications [1] describes a list of APIs\(^1\) that are exposed to the outside world. These APIs are the access points for external services to access information in the INFOD registry. The callers of these APIs send their request as an XML message (e.g., a SOAP message). After the request is handled, the APIs return a response message (e.g., a SOAP message) back to the caller. According to the specification [1], these APIs are categorized as follows:

<table>
<thead>
<tr>
<th>Managing Publishers</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>CreatePublisher</em></td>
</tr>
<tr>
<td><em>ReplacePublisher</em></td>
</tr>
<tr>
<td><em>DropPublisher</em></td>
</tr>
</tbody>
</table>

Table 1: Public APIs in Publisher Manager.

<table>
<thead>
<tr>
<th>Managing Subscribers</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>CreateSubscriber</em></td>
</tr>
<tr>
<td><em>ReplaceSubscriber</em></td>
</tr>
<tr>
<td><em>DropSubscriber</em></td>
</tr>
</tbody>
</table>

Table 2: Public APIs in Subscriber Manager.

Managing Consumers

\(^1\) API stands for Application Programming Interface. The APIs are the entry points to pass data into a system.
CreateConsumer
This operation creates a consumer entity in the INFOD registry.

ReplaceConsumer
This operation replaces the metadata associated with a particular consumer entity in the INFOD registry.

DropConsumer
This operation drops an existing consumer entity from the INFOD registry.

Table 3: Public APIs in Consumer Manager.

Managing Subscription

CreateSubscription
This operation creates a new subscription entity in the INFOD registry.

ReplaceSubscription
This operation replaces the metadata associated with a particular subscription entity in the INFOD registry.

DropSubscription
This operation drops an existing subscription entity from the INFOD registry.

Table 4: Public APIs in Subscription Manager.

Managing Vocabularies

RegisterPropertyVocabulary
This operation registers a property vocabulary to the INFOD registry.

CreatePropertyVocabularyInstance
This operation creates a new instance of a property vocabulary that is already registered in the INFOD registry.

DropPropertyVocabularyInstance
This operation drops an existing instance of a particular property vocabulary registered in the INFOD registry.

RegisterDataVocabulary
This operation registers a data vocabulary to the INFOD registry.

UnregisterVocabulary
This operation removes a vocabulary from the INFOD registry.

Table 5: Public APIs in Vocabulary Manager.

Managing Registry

GetMetaData
This operation queries the metadata of entities defined in the INFOD registry.

Table 6: Public APIs in Registry Manager.

The first step of applying aspect-oriented programming is to define the core and the aspects of the system being implemented. Since these APIs are the key requirements...
described in [1], they should always be accessible by the external services even though no aspect is woven into the system. Furthermore, these APIs are the entry points of the INFOD system and they do not crosscut the rest of the system. Therefore, they become the core of the INFOD system. Because these APIs are organized in six categories, each category is implemented as a manager in the core of the INFOD system. Each manager manages the APIs described in the associated category.

To find the aspects of the INFOD system, two approaches are used in this project. The first approach is described in [3]. The aspects of a system can be identified by analyzing the use cases of the system. The use cases are first defined from the requirements. Each use case is then described in details by categorizing its message flow in a base flow, an alternate flow and a sub-flow. To become an aspect, the requirements realized in the use case must satisfy the following criteria:

- The requirement is realized in either an alternate flow or a sub-flow of a use case.
- The requirement is crosscutting; i.e., the associated alternate flow or sub-flow is called by more than one use case of the system.

The use cases in [2] provide several examples of using the INFOD system on real business applications. These use cases have been used to come up with a set of generic use cases for the system. The overview of each generic use case is described in Appendix A.1. These use cases are further analyzed in Appendix A.2 to identify the

---

1 The base flow of a use case describes how data flows in a use case by default. It is being called directly by actors who want to access the system.
2 An alternate flow of a use case is an optional module branched out from the main module. It is being called when certain conditions become true.
3 A sub-flow of a use case is a module in the use case. The actors cannot call this type of module directly. It is called by the base flow or alternate flow of the same use case or those of other use cases.
crosscutting concerns and optional features that can be implemented as the aspects of the INFOD system. The result of this analysis is described in Section 3.3 INFOD System Aspects.

The second approach is derived from the horizontal decomposition principles discussed in one of the papers written by Charles Zhang [4]. This paper discusses some general guidelines to identify aspects from a software system:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Recognize the relativity of aspects. The semantics of an aspect is determined by the primary functionality of the application.</td>
<td></td>
</tr>
<tr>
<td>2. Establish the coherent core decomposition. The basis of aspect oriented decomposition is the establishment of a functionally coherent and vertically decomposed core.</td>
<td></td>
</tr>
<tr>
<td>3. Define the semantics of an aspect according to the core decomposition. Using the core as a reference, a functionality is considered orthogonal if both its semantics and its implementations are not local to a single component of the core.</td>
<td></td>
</tr>
<tr>
<td>4. Maintain a class-directional architecture. Crosscutting concerns should be implemented class-directional towards the core.</td>
<td></td>
</tr>
<tr>
<td>5. Apply incremental refactoring. Decomposition in the aspect dimension is assisted by incremental refactoring.</td>
<td></td>
</tr>
</tbody>
</table>

Table 7: This table lists the horizontal decomposition principles, adopted from [4].

The following criteria are derived from the five horizontal decomposition principles. To become an aspect, a feature must satisfy:

- The feature is addressing a crosscutting concern.
- The feature can be easy to replace. Developers can develop new versions of the feature without affecting the rest of the system.
- The feature is optional.

By using these two approaches, a list of potential aspects is identified for the INFOD system. The detail of each aspect is described in Section 3.3 INFOD System Aspects and
a comparison of the two approaches can be found in Section 7.4 Evaluation on the Approaches to Identify Aspects.

### 3.1 Core Architecture

Using the software development approach in [3], the core of the INFOD system has been identified from the generic use cases attached in Appendix A.1. It consists of six managers and each manager manages a specific area of interest. The class diagram below describes the overall structure of the INFOD core system.

![Class Diagram](image)

**Figure 2:** The class diagram describes the core of the INFOD system.

#### 3.1.1 Registry Manager

The Registry Manager contains the querying API described in [1]. The querying API allows the applications outside the INFOD system to access the content of the INFOD registry. This manager also provides a set of operations to update the INFOD registry so
that other managers in the system can call these operations to create, replace and remove an element from the registry. Furthermore, this manager examines the registered subscriptions when data is updated in the registry.

3.1.2 Publisher Manager

The Publisher Manager manages the publisher entities in the registry. It provides public APIs for creating, updating and removing publisher entities in the registry. If a publisher is notified to send a message to its eligible consumers, a message is generated for each notified publisher and sent to the consumers on behalf of the publishers.

3.1.3 Consumer Manager

The Consumer Manager manages the consumer entities in the registry. It provides public APIs for creating, updating and removing consumer entities in the registry. It is also responsible to propagate the data to the referenced service of the consumer entity when the entity receives a message from a publisher.

3.1.4 Subscriber Manager

The Subscriber Manager manages the subscriber entities in the registry. It provides public APIs for creating, updating and removing subscriber entities in the registry. A subscriber is mandatory to a subscription and the subscriber referenced by the subscription must exist in the registry before the subscription can be registered.

3.1.5 Subscription Manager

The Subscription Manager manages the subscription entities in the registry. It has operations for creating, updating and removing subscriptions in the registry. When a
subscription is updated in the INFOD registry, this manager will do a few things. First, this manager computes the consumers interested to receive data referenced by the subscription; then it notifies the eligible publishers to generate messages by sending a request to the publisher manager.

### 3.1.6 Vocabulary Manager

The Vocabulary Manager provides public APIs for creating, updating and removing vocabularies in the INFOD registry. This manager manages two types of user-defined vocabularies in the INFOD system: data vocabularies and property vocabularies. These vocabularies have to be defined by XML schemas. This manager also provides public APIs for creating and removing vocabulary instances in the INFOD registry. These vocabulary instances are instances of property vocabularies and they can be created only if the associated property vocabularies are already registered in the INFOD system.

### 3.1.7 Association Manager

The Association Manager manages the associations between the data vocabulary and the INFOD entities in the registry. It provides public APIs for creating and removing the vocabulary associations in the INFOD registry.

### 3.1.8 Utilities

The Utilities module provides a list of handy methods that can be called by all managers in the INFOD system. This module is also responsible to create all response messages returned by the public APIs specified in [1]. This module is internal to the INFOD system and thus outside systems do not know anything about this module.
3.1.9 XML Parser

The XML Parser module contains a list of operations to process data in XML format. This module is internal to the INFOD system and thus outside systems know nothing about this module. The parser can be used to process both XML messages and files in XML format.

3.2 Message Flow within the INFOD Core System

To give a better understanding of how the INFOD managers interact with each other within the INFOD core system, the following sequence diagrams illustrate the message flow in the system in several typical scenarios.

3.2.1 Creating a new Publisher Entity in the INFOD System

![Sequence diagram for creating a new publisher entity.]

Figure 3: Sequence diagram for creating a new publisher entity.

The sequence diagram above illustrates the message flow within the INFOD system when creating a new publisher entity.
When an actor (i.e., a service outside of the INFOD system) requests to register itself as a publisher, it first sends off a create publisher request to the INFOD system. When the request message arrives, the publisher manager of the INFOD system processes the request. It parses the request message and retrieves the publisher information from the message. Then it propagates the publisher information to the registry manager of the INFOD system. When the registry manager receives the data, it inserts the data into the registry and then generates a unique identifier (i.e., EPR\(^1\)) for the new publisher entity. The EPR is then returned to the publisher manager. When the publisher manager receives the EPR back from the registry manager, it wraps the EPR with a response message. The response message is then returned to the requested service.

### 3.2.2 Creating a new Subscription Entity in the INFOD System

![Figure 4: Sequence diagram for creating a new subscription entity.](image)

---

\(^1\) EPR stands for explicit reference. It is used to identify an entity registered in the INFOD system.
The sequence diagram above illustrates the message flow within the INFOD system when creating a new subscription entity.

When an actor (i.e., a subscriber) requests to add a new subscription to the INFOD registry, it first sends off a create subscription request to the INFOD system. When the request message arrives, the subscription manager of the INFOD system will process the request. It parses the request message and retrieves the subscription information from the message. Then it propagates the subscription information to the registry manager of the INFOD system. When the registry manager receives the subscription, it inserts the subscription into the registry and then generates an EPR for the new subscription entity. The EPR is then returned to the subscription manager. When the subscription manager receives the EPR back from the registry manager, it generates a notification message and sends it to the publisher manager to notify the publishers who are required to send messages for the data interested to this new subscription. When the publisher manager receives the notification, each notified publisher creates a notify message that contains the data interested to the new subscription. The publisher manager then finds the consumers referenced in the notification message sent by the subscription manager and sends the notify message to those consumers. After the subscription manager notifies the publisher manager, it wraps the returned EPR in a response message. The response message is then returned to the requested subscriber.
3.2.3 Updating Data Entries in the INFOD System

The sequence diagram above illustrates the message flow within the INFOD system when a user updates the data in the registry.

When an actor requests to update some data in the INFOD registry, it first sends off a data update request to the INFOD system. When the request arrives, the registry manager of the INFOD system processes the request. After the register manager finishes updating the registry, it examines the subscription entities in the INFOD system to see if it needs to notify the consumers who are interested to the changing data. When there is one or more active subscriptions, the subscription manager will be awakened and start processing them. The subscription manager generates a notification message for each active subscription and sends it to the publisher manager to notify the publishers who are
required to send messages for the data interested to the subscription. When the publisher manager receives the notification, each notified publisher creates a notify message that contains the interested data. The publisher manager then finds the consumers referenced in the subscription notification message and sends the notify message to those consumers. The subscription manager repeats this process until all active subscriptions are handled.

### 3.3 INFOD System Aspects

As the INFOD system is designed using the aspect-oriented approach, a number of aspects have been identified which can be woven into the core system. These aspects make the system more flexible and allow the system to have different versions of configurations when deploying it in the actual environment.

![Diagram illustrating the message flow between external services and the INFOD core system](image)
Figure 7: This diagram illustrates how the aspects intercept the message flow when dropping an INFOD entity from the INFOD registry.

Figure 8: This diagram illustrates how the logging and tracing aspect interacts with the INFOD core system.
Figure 9: This diagram illustrates how the message filtering aspect interacts with the INFOD core system.

The aspects shown in the diagrams above are described in more details below.

### 3.3.1 Attach SOAP Header Aspect

This aspect attaches a SOAP header to a given message. The given message must be in XML format. If the given message already has a SOAP header, this aspect does nothing. According to [1], all public APIs that manipulate the entities, vocabularies and associations in the INFOD registry require to return a response message to the requesting service. In addition, the response message must be wrapped in a SOAP envelope. By applying the criteria derived from [4], attaching a SOAP header to a message can be implemented as an aspect. In general, the main purpose of wrapping a message in a SOAP envelope is for sending the message in a remote environment. Since the APIs do not need the SOAP header to handle the request described by the message, the SOAP header becomes unnecessary. Since each API in the INFOD system returns a response
message after handling a request and these APIs are scattered across the INFOD system, attaching a SOAP header becomes a crosscutting concern. The following graph shows where this aspect is woven into the core files in terms of the line number in the code.

Figure 10: The line number at which the Attach SOAP Header aspect is woven into the core.

Furthermore, SOAP is not the only protocol that can be used to transport a message from one place to another. If the system wants to switch to a different protocol or requires the update of the SOAP version to exchange messages, implementing this as an aspect can also allow all the changes to be done in one place. Since this concern satisfies all the criteria derived from [4], it becomes an excellent aspect to implement in the INFOD system.

### 3.3.2 Remove SOAP Header Aspect

This aspect removes a SOAP header from a given message. If the given message does not contain a SOAP header, this aspect does nothing. According to [1], all public APIs that manipulate the entities, vocabularies and associations in the INFOD registry require its request message to be wrapped in a SOAP envelope. By applying the criteria
described in [4], removing the SOAP header from a message can be implemented as an aspect. The SOAP header is only for transporting messages from one place to another. After the INFOD system picks up the messages, the SOAP header of the message is no longer needed. Although the INFOD system can still process the message with the SOAP header, removing the header can help the INFOD core system to allocate less memory to handle the request. Furthermore, SOAP is not the only protocol that can be used to transport a message from one place to another. If the system wants to switch to a different protocol or requires the update of the SOAP version to exchange messages, implementing this as an aspect allows all the changes to be done in one place. Also, most importantly, it is addressing a crosscutting concern because the requests are being handled in different places of the INFOD system depending on the request type. The crosscutting concern can be illustrated by the following graph which shows where this aspect is woven into the core files.

![Graph](image)

Figure 11: The line number at where the Remove SOAP Header aspect is woven into the core.
Since this concern satisfies all the criteria described in [4], it becomes an excellent aspect to implement in the INFOD system.

Although this aspect and the Attach SOAP header aspect are both dealing with SOAP headers, it is better to have two separate aspects rather than grouping them into one. The reason is that it is not necessary to have both the request and response messages wrapped in SOAP. If both of them are in the same aspect, it becomes a limitation to the INFOD system if it wants to use different protocols when receiving and sending messages. Thus, the Attach SOAP header and Remove SOAP header should be two independent aspects.

### 3.3.3 Message Filtering Aspect

In the INFOD system, filtering can be done by specifying property constraints in the entities, vocabularies and associations. These property constraints can tell how these INFOD elements interact with each other. Using the approach mentioned in [3], the message filtering can be implemented as an aspect. First, the message filtering requirement is realized in a separate module of the use case. For more details of the use case realizing the message filtering, please go to the Use Case Diagram in Appendix A.1. It is a sub-flow module because the actors should never access it directly. The message filtering is also a crosscutting concern because it is being called by multiple use cases: the use case of creating a subscription and the use case of replacing a subscription. Therefore, it is a good idea to implement it in a common place so that it will be easy to update or replace. This crosscutting pattern can be illustrated in the following graph.
Furthermore, the message filtering can be seen as an optional concern. If it is taken away from the core system, the core can still work without any problem. Because the message filtering satisfies all the criteria of the approach described in [3], it becomes a good candidate of aspect to implement in the INFOD system.

### 3.3.4 Drop Unused INFOD Elements Aspect

The INFOD system provides a list of public APIs to remove entities, vocabularies and associations from the registry. These APIs are declared in most of the managers in the core system. These drop operations provide several options to remove an existing element from the registry. By default, the system removes the specified element and other elements that reference the removing element. This aspect provides another option to remove an INFOD element. It removes the specified INFOD element only if the element is not referenced by other elements in the system. Using the approach in [3], this requirement can be implemented as an aspect. It is realized as a separate module in the alternate flow of the use case because this requirement is an alternate way of removing
entities, vocabularies and associations from the registry. According to the specification [1], all drop operations has default behaviour which is already implemented in the core. Therefore, the INFOD core system will not break if this requirement is taken away. Furthermore, this remove option is a crosscutting concern because it is used in all use cases that perform the drop operation in the core system and thus it is a good idea to implement it in a common place so that it is easier to update or replace. The following graph shows where this aspect is woven into the core files.

Figure 13: The line number at where the Drop Unused INFOD Elements aspect woven into the core.

Since it meets all the criteria outlined by the approach in [3], the **Drop Unused INFOD Elements** become an excellent candidate of aspect to implement in the INFOD system.

### 3.3.5 Disable New INFOD Elements Aspect

The INFOD system provides a third option to remove an entity, vocabulary or association from the registry. By using a reference count, the system knows how many INFOD elements are currently referencing the element that wants to be removed. This aspect allows the system to disable any new reference to the element to be removed and the
element will be removed only if the reference count of the removing element reaches zero.

By using the approach in [3], this requirement can be implemented as an aspect. It is realized as a separate module in the alternate flow of the use case because this requirement is an alternate way to remove entities, vocabularies and associations from the registry. Since there is a default behaviour already implemented in the core system, the INFOD core system will still work without any problem if this requirement is taken away. Furthermore, this remove option is a crosscutting concern because it is used in all use cases that perform the drop operation in the core system and thus it is a good idea to implement it in a common place so that it is easier to update or replace. The following graph shows where this aspect is woven into the core files.

![Graph showing line numbers for various core files]

Figure 14: The line number at where the Disable New INFOD Elements aspect is woven into the core.

Since it meets all the criteria outlined by the approach in [3], the **Disable New INFOD Elements** becomes an excellent candidate of aspect to implement in the INFOD system.
3.3.6 Manipulate Registry Aspect

Regardless of whether the INFOD system is performing a creation, modification or deletion of an INFOD element, it is always necessary to update the registry to reflect the changes. Manipulating the INFOD registry is a crosscutting concern because almost every operation in the INFOD system needs to deal with the registry. Therefore, it becomes a potential candidate of an aspect. Although it is mandatory to manipulate the registry, by applying the criteria described in [4], it may still be a good aspect to implement. First, this requirement is a crosscutting concern because every manager in the INFOD core system needs to access the registry when handling requests. Therefore, it becomes a good idea to implement this requirement in a separate module so that all manipulations to the INFOD registry are done in one common place. Second, the INFOD registry is implemented as a metadata file in this project. Implementing this requirement as an aspect provides a flexible option for the INFOD system to switch to a different database platform (e.g., DB2, Oracle, etc.) in the future and thus it becomes much easier to switch to a different database system without modifying the core. Since this requirement satisfies the criteria described in [4], it can be a good candidate of aspect to implement in the system.

However, it becomes apparent that this feature cannot be implemented as an aspect during the implementation phase. The reason is that an aspect can be taken out from the core system anytime. Manipulating the registry is mandatory to the INFOD system because every request requires either a read or write to the registry to get or update information registered in the INFOD system. Therefore, the INFOD system will not
function if this feature is missing in the system. As a result, this feature is implemented as part of the Registry Manager.

### 3.3.7 Authorization Failure Aspect

To implement secured access to the INFOD registry, one of the ways is to limit only a certain users or services to access to the registry. When a user or service is not authorized to access the registry, the operation processing the request returns a fault message to the requesting user or service instead of a normal response message. By applying the criteria described in [4], this requirement can be implemented as an aspect. The authorization process is a crosscutting concern because the authorization is done in several places in the core system when creating, replacing or dropping an element in the INFOD registry. This crosscutting behaviour is illustrated in the following graph.

![Figure 15: The line number at where the Authorization Failure aspect is woven into the core.](image)

Furthermore, implementing authorization failures as an aspect gives the INFOD system a flexible option for upgrade in the future. For example, if there is a new public API to be added to the INFOD system and it can throw a new type of authorization failure message,
it becomes much easier to adapt to the new change because all database authorization failures are implemented in one common place. Also, since it is part of the error handling, it is not mandatory to the core system. Since **Authorization Failure** satisfies all the criteria described in [4], it becomes an excellent candidate of aspect to implement in the INFOD system.

### 3.3.8 Error Handling Aspect

Since a request message may contain invalid information, it is necessary to validate the request message before processing the request. The error validation is implemented as an aspect. According to [1], all public APIs declared in the INFOD system take a request message as their input parameter and thus the error validation to a request message becomes a crosscutting concern. This can be realized in the use cases described in Appendix A.1. Implementing the error validation in the same module allows the developers to have a consolidated view on all the error handling routines of the whole INFOD system. The following graph illustrates the crosscutting behaviour of this aspect.

![Figure 16: The line number at where the Exception Handling aspect is woven into the core.](image-url)
By using the approach in [3], error handling always falls in the *alternate flow* of a use case which implies that the error handling is always not mandatory to the core system. Since the error handling requirement satisfies all the criteria outlined in the approach in [3], it becomes an excellent aspect to implement in the INFOD system.

Although both *Error Handling* aspect and the *Authorization Failure* aspect are being used to handle error conditions, it is better to make them two independent aspects. It is because the targets to be validated by these two aspects are totally different. The *Authorization Failure* aspect looks for errors when updating the INFOD registry and thus it will likely be updated when the platform of the INFOD registry is changed. On the other hand, the *Error Handling* aspect looks for errors in a request message and thus it will likely be updated when there are major changes to the APIs that take the request message.

### 3.3.9 Logging and Tracing Aspect

Logging and tracing is traditionally an excellent aspect to implement. It is used mainly to provide debugging information to the developers when implementing the system. By applying the criteria derived from [4], it is proved to be an excellent aspect to implement. First, it is an optional feature because the main purpose of this aspect is to log the call stack to provide debugging information. Therefore, the INFOD core system can always run without this aspect. Second, it is crosscutting because it can scatter the trace statements all over the system. Third, logging and tracing can be changed frequently to adapt to different debugging purposes and thus implementing it as an aspect allows developers to make frequent changes without impacting the core system. Since it
satisfies all the criteria, logging and tracing becomes an excellent candidate of aspect to implement in the INFOD system. Note that the **Logging and Tracing** aspect is internal to the INFOD core system and thus outside systems should know nothing about this aspect. The following graph illustrates the crosscutting behaviour of this aspect.

![Diagram](image)

*Figure 17: The line number at where the Logging and Tracing aspect is woven into the core.*

### 3.3.10 Remove INFOD References Aspect

This aspect is realized when implementing the core system. When removing an INFOD entity, vocabulary, vocabulary instance or association, the default behaviour is to also recursively remove all the INFOD elements references the removing INFOD element. By applying the criteria derived from [4], the removal of these references can be implemented as an aspect. First, it is a crosscutting concern because this is done in every drop operation implemented in every manager. Therefore, handling the removal of these references in one common module allows developers to debug and update the module without affecting the rest of the system. This crosscutting behaviour is illustrated in the following graph.
Second, this requirement is optional. If the core system does not remove the referencing INFOD elements, the only disadvantage is that the INFOD registry will contain dangling references to the removed element but it will not break the system. Since it satisfies the criteria derived from [4], it becomes a good candidate of aspect to implement in the INFOD system.

### 3.3.11 Remove Namespace Aspect

This aspect is realized when implementing the core system. The purpose of this aspect is to remove the namespace information found in the request message. By applying the criteria described in [4], it can be implemented as an aspect. First, removing the namespace information in the message is optional. Each message handled by the core system originally has namespace attributes to identify where the message elements are defined. However, these namespace attributes are not needed by the core system. Therefore, removing these namespace attributes before handling the messages does not have any impact to the core system. Also, if these namespace attributes are removed
from the message, the core system does not have to worry about them and it can help the core system to reduce the memory allocation to cache the message when processing it.

Second, removing the namespace information from the messages is a crosscutting concern because all the public APIs implemented in the core managers takes a request message as their input parameter. Therefore, the namespace information should be removed from these request messages before the managers handle them. This crosscutting behaviour is illustrated in the following graph.

![Graph](image)

Figure 19: The line number at which the Remove Namespace aspect is woven into the core.

Third, implementing this requirement in one common module allows developers to easily update the implementation in the future. Since this requirement meets all the criteria, it becomes an excellent candidate of aspect to implement in the INFOD system.

### 3.4 External Services

By analyzing the use cases described in Appendix A.1, the following external services are required to implement the INFOD system.
3.4.1 XML Parser

Since all request and response messages exchanging in the system are in XML format, an XML parser is required to implement this system. *Libxml2* is used for implementing the INFOD system.

*Libxml2* is the XML C parser and toolkit developed for the Gnome project. It is a free software available under the MIT License. *Libxml2* is known to be very portable, the library is written in C and it should build and work without serious troubles on a variety of systems (Linux, Unix, Windows, CygWin, MacOS, MacOS X, RISC Os, OS/2, etc.). *Libxml2* implements a number of existing standards related to markup languages. For a list of these standards, please refer to Appendix A.6.

3.4.2 XQuery Parser

According to [1], all property constraints and querying are expressed in XQuery expressions. To simplify the implementation of the INFOD system, they are changed to XPath expressions. Since libxml2 already fully implemented XPath language, libxml2 is used to process XPath expressions. The following are some examples of XPath expressions:

- //infodPublisher – Selects all INFOD publisher entities in the INFOD registry.
- //infodCustomer[@epr='http://www.ggf.com/SomeCustomerEPR'] – Selects the INFOD customer entity with an EPR equals
- `//infodSubscriber[SubscriberName = 'John Smith']` – Selects the INFOD subscriber entities that have a SubscriberName element with a string value of “John Smith”.

### 3.4.3 Database

The INFOD system is using metadata in XML format to store the vocabulary structures and the vocabulary instances. They are stored in a file. When processing a request, the INFOD registry loads the metadata file and checks if the elements referenced in the request message can be found. Then the INFOD registry updates the file if needed to complete the request.

Since the INFOD registry is using metadata to store its data, it can be deployed on any platform and does not need to worry about having a different version of database among various platforms.

Below is an example of the content in a metadata file:

```xml
<publisher>
  <infopublisher>
    <WSEntityReference>wsa:http://www.carcommunity.com/CarDealerServices</WSEntityReference>
    <PublisherName>Frontier Ford</PublisherName>
    <PublisherDescription>Oldest Ford Dealer in SFO Bay Area</PublisherDescription>
    <PropertyConstraints>XQuery Expression</PropertyConstraints>
  </infopublisher>
</publisher>

<subscriber>
  <infosubscriber>
    <WSEntityReference>wsa:http://www.carcommunity.com/CarBuyerEmail</WSEntityReference>
    <SubscriberName>Susan Maria Callas</SubscriberName>
  </infosubscriber>
</subscriber>
```
3.5 **Non-Functional Requirements**

3.5.1 Remote Access to INFOD System

The following diagram shows how to remotely access the INFOD system.

![Remote Access Diagram](image)

**Figure 20: Remote access to INFOD system.**

With the use of C sockets, it becomes possible to deploy and run the INFOD system in a distributed environment. Instead of calling the public APIs of the INFOD system directly, other services and systems drop their request messages to the C socket channel and the INFOD system then picks up the messages from the C socket channel on the other side. After a request message is picked up, the INFOD system determines the type of the
request and delegates the request to the right INFOD manager to handle. Please refer to
Section 3.1 for details of each manager implemented for the INFOD system.

In this project, since all the test cases are originally designed to access the INFOD system
by direct procedural calls, there is an aspect on the client side to drop the request
messages created by the test cases to the socket channel. Similarly, on the server side,
there is a new module which picks up the request messages from the socket channel and
directs the messages to the right INFOD manager to handle the request. The following
graph illustrates the crosscutting behavior of the aspect on the client side.

![Graph](image)

Figure 21: The line number at which the Client Socket aspect is woven into the core.

3.5.2 High Performance

The following diagram describes how interprocess communication is done in the INFOD
system to achieve high performance.
With the use of interprocess communication in C, the INFOD system can achieve higher performance by spawning a separate child process to handle each service request. When the INFOD system receives a new request from a Web service, it determines the type of the request. Before delegating the request to the particular manager, the INFOD system spawns a new child process for the particular manager to handle the request. Within the child process, the manager has its own resources which cannot be touched and interrupted by other managers who are handling other requests. However, since the INFOD registry is the data centre of the INFOD system and all managers need to access the registry to handle the request, the registry is shared by all the child processes. A simple algorithm is used to keep track of who is currently accessing the registry. When a process is currently accessing the registry, the algorithm blocks other processes from accessing the registry so that the data in the registry is not corrupted. Once the process is done with the registry, the lock is released so that the next process in the waiting queue can access the registry.
This feature is implemented as an aspect because of three reasons. First, this feature is an optional feature; the INFOD system can still work properly without this feature. Second, this feature is crosscutting. There are several places in the code that need to be intercepted to spawn a new child process and access the shared registry of the INFOD system. The following graph illustrates this crosscutting behavior among the core files.

![Figure 23: The line number at which the High Performance aspect is woven into the core.](image)

Third, all the code for interprocess communication is gathered in one central place for update or upgrade in the future. Since it satisfies all the criteria derived from [4], it becomes another excellent candidate of aspect to implement in the INFOD system.
4 Implementation

The following table shows a summary of the aspects that are identified using the two approaches described in the previous chapter. The implementation details of these aspects are further discussed in the next sections.

<table>
<thead>
<tr>
<th>Approach Applied</th>
<th>Aspects Identified</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Case Scenario Approach</td>
<td>Message Filtering</td>
</tr>
<tr>
<td></td>
<td>Drop Unused INFOD Elements</td>
</tr>
<tr>
<td></td>
<td>Disable New INFOD Elements</td>
</tr>
<tr>
<td></td>
<td>Error Handling</td>
</tr>
<tr>
<td>Horizontal Decomposition Principles Approach</td>
<td>Attach SOAP Header</td>
</tr>
<tr>
<td></td>
<td>Remove SOAP Header</td>
</tr>
<tr>
<td></td>
<td>Authorization Failure</td>
</tr>
<tr>
<td></td>
<td>Logging and Tracing</td>
</tr>
<tr>
<td></td>
<td>Remove INFOD References</td>
</tr>
<tr>
<td></td>
<td>Remove Namespace</td>
</tr>
<tr>
<td></td>
<td>Client Socket</td>
</tr>
<tr>
<td></td>
<td>High Performance</td>
</tr>
</tbody>
</table>

Table 8: The table summarizes the aspects identified in the INFOD system.

4.1 Attach SOAP Header Aspect

The main purpose of this aspect is to attach a SOAP header to a returning message. This aspect is a domain specific aspect because it is implemented specifically for attaching a SOAP header to a message in the INFOD system. Since other systems may not deal with messages or their messages can be in different format (e.g., non-XML), this aspect is less likely reusable in other systems.

In this aspect, there are three around advices being woven into the execution of the INFOD core system.
The code snippet above is one of the advices found in the **Attach SOAP Header** aspect. This advice is woven into the execution of `create$ResponseMessage(...)`. When the INFOD core system calls the `create$ResponseMessage(...)` in the **Utility Manager**, this advice comes to play. It checks to see if the returning message is already wrapped in a SOAP envelope. If not, the aspect creates a SOAP envelope and puts the message in the SOAP envelope.

```c
message around(): execution($ create$ResponseMessage(...))
{
  xmlDocPtr bodyDoc;
  xmlNodePtr bodyRootNode;
  xmlDocPtr soapDoc;
  xmlChar * result;
  int size;
  message response:

  response = proceed();
  bodyDoc = parse(response);
  bodyRootNode = xmlDocGetRootElement(bodyDoc);

  soapDoc = createSoapMessage(NULL, bodyRootNode);
  xmlDocDumpFormatMemory(soapDoc, &result, &size, 1);
  return result;
}
```

The code snippet above is the second advice found in the **Attach SOAP Header** aspect. Similar to the first advice, this is an **around** advice and it is woven into the execution of `createSubscriptionNotification(...)`. The `createSubscriptionNotification(...)` creates a subscription notification message to notify the publishers to send messages to the consumers specified in the subscription just handled by the **Subscription Manager**. The advice first allows the `createSubscriptionNotification(...)` to proceed. Then it checks if
the returning response message is a SOAP message. If not, it creates a SOAP envelope and puts the returning message in the SOAP envelope.

```c
message around(): execution[& createMessage(...)]
{
    xmlDocPtr bodyDoc;
    xmlNodePtr bodyRootNode;
    xmlDocPtr soapDoc;
    xmlChar * result;
    int size;
    message response;

    response = proceed();
    bodyDoc = parse(response);
    bodyRootNode = xmlDocGetRootElement(bodyDoc);

    soapDoc = createSoapMessage("http://www.ggf.org/infod/INFODNotify/Notify", bodyRootNode);
    xmlDocDumpFormatMemory(soapDoc, result, &size, 1);
    return result;
}
```

The code snippet above shows the third advice in the **Attach SOAP Header** aspect. It is an *around* advice and it is woven into the execution of the `createMessage(...)` in the **Publisher Manager**. Similar to the previous two advices, it checks if the returning message is already a SOAP message. If not, it creates a SOAP envelope and puts the returning message in the SOAP envelope.

### 4.2 Remove SOAP Header Aspect

The main purpose of this aspect is to remove any SOAP header attached to a given message. This aspect is a domain specific aspect because it is implemented specifically for removing a SOAP header from a message in the INFOD system. Since other systems may not deal with messages or their messages can be in different format (e.g., non-XML), this aspect is less likely reusable in other systems.
The code snippet above is the advices found in the **Remove SOAP Header** aspect. All these advices are *before* advices to the public APIs described in [1]. These advices are woven into the INFOD core system before the execution of the public APIs in the INFOD core system. When these advices come to play, they first check if their given message is a SOAP message. If so, it removes the SOAP header from the message before the core functions handle these messages.

### 4.3 Message Filtering Aspect

The main purpose of this aspect is to filter the messages sent from publishers to consumers. It filters the messages by examining the property constraints\(^1\) of the following INFOD elements:

- **Entities** (i.e., Publishers, Consumers, Subscribers, Subscriptions)

---

\(^1\) A constraint is used to specify which instances of data are of interest to an entity. A constraint can reference single instances or a set of instances, which may represent entities, property vocabulary instances, messages, events or state transitions.
• Associations

• Property Vocabulary Instances

This aspect is a domain specific aspect because it is implemented specifically for message filtering in the INFOD system. The property constraints and the registry elements that it analyzes are unique to the INFOD system. Therefore, it is impossible to reuse this aspect in other systems.

```c
void around(char * consumerAddr, message msg); call(around processdata(char *, message) & args(consumerAddr, msg))
{
  xmlDocPtr doc;
  xmlXPathObjectPtr result;
  char * subscriptionEPR;
  char * publisherEPR;
  int size;
  xmlNodePtr node;
  xmlNodePtr consumerNode;
  xmlNodePtr constraintNode;
  int i;
  int isFilter;
  char * xpathExpr;

doc = parseDoc(
  result = searchDoc(doc, "//wsp:SubcriberReference/wsa:Address");
  if (size > 0)
  {
    node = result->nodesetval->nodeTab[0];
    subscriptionEPR = xpathEval(doc, "//wsp:SubscriptionEPR[@wsp:Location]/text()");
    publisherEPR = xpathEval(doc, "//wsp:PublisherEPR[@wsp:Location]/text()");
    constraintNode = xpathEval(doc, "//wsp:Consumer[@wsp:Location]/text()");
    if (isFilter = hasElem(xpathExpr, subscriptionEPR))
      if (isFilter == 0)
        return;
    if (isFilter = hasElem(xpathExpr, publisherEPR))
    {
      return;
    }
    if (isFilter = hasElem(xpathExpr, constraintNode))
    {
      return;
    }
  }

  isFilter = hasElem(xpathExpr, consumerNode);
  if (isFilter == 0)
    return;
}
```
The code snippet above is an *around* advice in the **Message Filtering** aspect that is woven into the execution of `processData(…)` in the **Consumer Manager**. The purpose of this advice is to filter the message when the **Consumer Manager** receives a message from the **Publisher Manager**. The advice checks if the given message should be propagated to the consumer by checking the property constraints in the associated consumer entity in the INFOD registry. If it should not, the advice drops the message so that the consumer never receives the message.

```csharp
char * around(xmlDocPtr notification, xmlNodePtr entityRef): call(char * getPublisherEPR(xmlDocPtr, xmlNodePtr)) &
args(notification, entityRef) & Infunc(computePublisherList)
message around(char * consumerEPR, char * subscriptionEPR, char * publisherEPR, char * vocabEPR):
call(message createMessage(char *, char *, char *, char *, xmlNodePtr)) &
args(consumerEPR, subscriptionEPR, publisherEPR, vocabEPR, xmlNodePtr)
xmlNodePtr around(xmlNodePtr ns, const char * consumerConstraintsExpr, char * subscriptionExpr):
call(xmlNodePtr computeStaticConsumers(xmlNodePtr, const char *, char *)) &
args(ns, consumerConstraintsExpr, subscriptionExpr)
```

The code snippet above shows the rest of the advices found in the **Message Filtering** aspect. The first advice in the code snippet is woven into the execution of computing the publisher list in the **Publisher Manager**. By examining the property constraints in the current handling subscription and its referenced subscriber, it filters which publisher is eligible to publish messages to the consumer. The second advice is woven into the execution of `createMessage(…)` in the **Publisher Manager**. By examining the property constraints in corresponding INFOD associations and INFOD publishers, it determines which publisher is eligible to publish message for the current handling subscription.

Lastly, the third advice is woven into the execution of `computeStaticConsumers(…)` in the **Subscription Manager**. By examining the property constraints in the subscription and its referenced subscriber, it determines which consumer is eligible to receive the data specified in the subscription. If any of these advices fails to comply with the constraints
specified in the INFOD elements, the request for publishing the subscription data is ignored.

### 4.4 Drop Unused INFOD Elements Aspect

The main purpose of this aspect is to determine if a given element is referenced by other elements in the INFOD registry. If the element is not referenced, the core method will proceed to remove the element from the registry. Otherwise, it returns the “COMPLETE” status to the caller to signal that the removal of the INFOD element is successful.

This aspect is a domain specific aspect because it is implemented specifically for the execution mode **IF UNUSED** that is described in the specification. In addition, it accesses the INFOD registry directly to determine if the given element is referenced in the registry. Therefore, this aspect cannot run without the INFOD registry. As a result, it is impossible to reuse this aspect in other systems.

```c
const char* around(const char* xpath, char* eprvalue, char* mode)
{
    call(const char* dropregistry(const char*, char*, char*))
    && infunc(dropSubscriber)
    && args(xpath, eprvalue, mode)
    {
        if (strcmp(mode, "DISABLE NEW") == 0) || (strcmp(mode, "CASCADE") == 0))
        {
            return proceed();
        }
        else
        {
            if (!hasEntityReference(eprvalue, mode) && !hasVocabularyReference(eprvalue, mode))
            {
                return proceed();
            }
            else
            {
                return "COMPLETED";
            }
        }
    }

    const char* around(const char* xpath, char* eprvalue, char* mode);
    call(const char* dropregistry(const char*, char*, char*))
    && infunc(dropPublisher)
    && args(xpath, eprvalue, mode)
    {
        if (strcmp(mode, "DISABLE NEW") == 0) || (strcmp(mode, "CASCADE") == 0))
        {
            return proceed();
        }
        else
        {
            if (!hasEntityReference(eprvalue, mode) && !hasVocabularyReference(eprvalue, mode))
            {
                return proceed();
            }
            else
            {
                return "COMPLETED";
            }
        }
    }
}
The above code snippets shows the *around* advices in the **Drop Unused INFOD Element** aspect. These advices are activated when the core system calls the `dropRegistry(...)` in the **Registry Manager**. They cannot be consolidated into one advice because the advice needs to know the type of the removing element to determine where to look for the referencing INFOD elements. For example, a data vocabulary can only be referenced by an association; a subscriber can only be referenced by a subscription.

Since the `dropRegistry(...)` does not care about the type of the removing element, the only way of knowing its type is to check who calls `dropRegistry(...)`. These advices allow `dropRegistry(...)` to proceed if the execution mode is not “IF UNUSED” or the removing INFOD element is not referenced by any other INFOD elements in the registry.

Otherwise, it simply returns a “COMPLETE” status to the caller.
4.5 Disable New INFOD Elements Aspect

The main purpose of this aspect is to determine if a given element should be removed from the INFOD registry. If the element is not referenced, the core method will proceed to remove the element from the registry. Otherwise, it sets the "DISABLE" flag of the element. Any new INFOD element can no longer reference this element and this element is removed only when other INFOD elements no longer reference it.

This aspect is a domain specific aspect because it is implemented specifically for the execution mode "DISABLE NEW" that is described in the specification. In addition, it accesses the INFOD registry directly to determine if the given element is referenced in the registry. Therefore, this aspect cannot run without the INFOD registry. As a result, it is impossible to reuse this aspect in other systems.
The above code snippets shows the around advices in the **Disable New INFOD Element** aspect. These advices are activated when the core system calls the `dropRegistry(...)` in the **Registry Manager**. Similar to the advices in **Drop Unused INFOD Element** aspect, they cannot be consolidated into one advice because the advice needs to know the type of the removing element before determining their referencing INFOD elements. These advices allow `dropRegistry(...)` to proceed if the execution mode is not “DISABLE NEW” or the removing INFOD element is not referenced by any other INFOD elements in the registry. Otherwise, it sets the “DISABLE” flag of the removing element and then returns a “COMPLETE” status to the caller.

```cpp
const char * around(const char * xpath, char * eprvalue, char * mode);
call(const char * dropRegister(const char *, char *, char *)) && infunc(dropPublisher) && args(xpath, eprvalue, mode);
call(const char * dropRegister(const char *, char *, char *)) && infunc(dropConsumer) && args(xpath, eprvalue, mode);
call(const char * dropRegister(const char *, char *, char *)) && infunc(unregistervocabulary) && args(xpath, eprvalue, mode)
```

Besides the around advices, the above code snippet shows an after advice in the **Disable New INFOD Element** aspect. This advice is woven into the call of `dropRegistry(...)` and is executed after the call is done. The purpose of this advice is to check if there is any disabled INFOD element that can be removed after an INFOD element is removed. It is done this way because the INFOD element just removed from the system may be the last
element referenced the disabled elements. If the disable elements are no longer being referenced, the aspect calls the \texttt{dropRegistry(…)} in the \textit{Registry Manager} to remove them from the registry.

\section*{4.6 Authorization Failure Aspect}

The purpose of this aspect is to determine if the user who requests the service is authorized to access the INFOD registry. If the user does, the aspect lets the core system to handle the request. Otherwise, the aspect returns an authorization exception to the user. This aspect is a generic aspect because it can be isolated from the INFOD system. The user information is in the message header and the username database is in a separate file in the system. Therefore, the authorization check can be done outside of the INFOD system. Since the username database and the authorization check can be moved easily from one system to another, it should be fairly easy to port this aspect to another system.

Since there are other aspects that manipulate the message header, the authorization check is done in two steps.

\begin{verbatim}
before(message request): execution(message create$(message)) && args(request)
{    authorized = isAuthorized(request);
}
before(message request): execution(message replace$(message)) && args(request)
{    authorized = isAuthorized(request);
}
before(message request): execution(message drop$(message)) && args(request)
{    authorized = isAuthorized(request);
}
before(message request): execution(message register$(message)) && args(request)
{    authorized = isAuthorized(request);
}
\end{verbatim}
The above code snippet shows the before advices in the Authorization Failure aspect.

These advices are woven into the execution of the public APIs (CreatePublisher, ReplaceConsumer, DropSubscriber, etc) described in the specification [1]. They do the authorization check from the header of the given message and put the result in a local variable in the aspect.

The second step of the authorization check is done in the around advices. Similarly, these advices are woven into the execution of each public APIs described in the specification [1]. When the request messages get to these around advices, their message header is already removed by Remove SOAP Header aspect. Therefore, the advices checks if the local variable authorized is true. If it is true, it means the user requesting the service is authorized to access the INFOD registry and thus it will allow the core system to handle the request. Otherwise, these advices create an authorization exception and return it to the user.
4.7 Error Handling Aspect

The purpose of this aspect is to handle error conditions in the INFOD system. When a request message comes into the system, it may contain invalid information. This aspect makes sure that all request messages are valid and that they can be handled properly by the INFOD system.

Although this aspect sounds like it is a global aspect which can be reused easily by other systems, it is actually a domain specific aspect. This is because this aspect checks for errors that only apply to the INFOD system. For example, it checks if an element referenced in the request message already exists in the INFOD registry. This reference check is only valid in INFOD system. Therefore, this aspect cannot be reused by other systems.

```c
message around(message request): execution(message getMetaData(message)) & args(request)
{
  xmlDocPtr doc;
  xmlNodePtr result;
  xmlDocPtr node;
  int size;
  char * mode;
  char * faultDescription;

  doc = parse(request);
  result = searchDoc(doc, "//MetadataQueryExpression");
  size = (result->nodeSetVal) ? result->nodeSetVal->nodeNr : 0;
  if (size == 0)
    faultDescription = "The request message is missing the MetadataQueryExpression attribute.\n";
    return createFault("MissingRequiredParameterFault", faultDescription);
  return proceed();
}

message around(message request): execution(message registerDataVocabulary(message)) & args(request)
{
  xmlDocPtr doc;
  xmlPathObjectPtr result;
  xmlDocPtr node;
  int size;
  char * mode;
  char * faultDescription;

  doc = parse(request);
  result = searchDoc(doc, "//VocabularyLanguage");
  size = (result->nodeSetVal) ? result->nodeSetVal->nodeNr : 0;
  if (size == 0)
    faultDescription = "The request message is missing the VocabularyLanguage attribute.\n";
    return createFault("MissingRequiredParameterFault", faultDescription);
  return proceed();
}
The code snippets above are some of the advices found in the Error Handling aspect.

All of them are around advices so that they can return the caller with an exception message if the request message is invalid. These advices are woven into the execution of the public APIs (CreatePublisher, CreateConsumer, etc) described in the specification [1]. The error checks are done differently depending on the type of the request message. If there is an error found in the request message, the advice returns an exception message describing the error. Otherwise, it allows the core system to proceed and handle the request.

4.8 Logging and Tracing Aspect

The purpose of this aspect is to provide logging and tracing capabilities to the core system to help the developers to implement and debug the INFOD system. The Logging
and Tracing is always a global aspect. Since it is not depending on the INFOD system, it is very easy to port this aspect to another system.

```c
before(): execution(message create$(message)) || execution(message replace$(message)) || execution(message drop$(message)) || execution(message $register$(message))
    { printf("Before entering function: %s\n", this->funcName);
    }

before(): execution(message $associate$(message))
    { printf("Before entering function: %s\n", this->funcName);
    }

before(message value, char * subscriptionEPR): execution($ sendMessage(char *, message)) &&
    args(subscriptionEPR, value)
    { printf("Subscription notification created for %s\n", subscriptionEPR);
      printf("%s\n", value);
    }
```

The code snippets above show the advices in the **Logging and Tracing** aspect. They are woven into the execution of the core system and print debugging information to the console. For example, the last *before* advice is activated before the `sendMessage(…)` method is executed. It prints the *subscription EPR* and the *subscription notification* message created for a specific subscription.

### 4.9 Remove INFOD References Aspect

The main purpose of this aspect is to remove all INFOD elements that reference the just deleted element from the INFOD registry. Since only a certain type of elements can be referenced, the advices determine the type of the removed element from the call stack and search for their referencing elements in the registry. If the aspect finds any referencing elements, they are also removed from the registry.

This aspect is a domain specific aspect because it requires the INFOD registry to find the referencing INFOD elements. Thus, it cannot be isolated from the INFOD system and so it is not possible to reuse this aspect in another system.
The above code snippet shows the before advices in the **Remove INFOD References** aspect. They are activated when the `dropRegistry(...)` in the **Registry Manager** is called.

They cannot be consolidated into one advice because the only way to know the type of the removing INFOD element is from the call stack. Since each INFOD element type can only be referenced in certain types of INFOD elements, the advices use the type of the removing element to determine which INFOD elements they should look for. For example, if the removing element is an INFOD publisher, the advice only looks for references in INFOD associations and INFOD vocabulary instances. If the advices find other INFOD elements reference the removed element, they remove them from the registry as well.
4.10 Remove Namespace Aspect

The main purpose of this aspect is to remove the namespace information from a request message. Since the INFOD core system does not need the namespace information in the message to handle the request, the namespace information can be removed so that the core system does not need to worry about them.

This aspect is a generic aspect because it can be isolated from the INFOD system easily. All the work is done in the given request message. Therefore, it can easily be reused in other systems. Because the Remove SOAP Header aspect needs the namespace information to locate the SOAP header, this aspect must be executed after the Remove SOAP Header aspect.

```java
before(message $ request): execution(message create$(message)) && args(*request) {
    removeNamespace(request);
}
before(message $ request): execution(message replace$(message)) && args(*request) {
    removeNamespace(request);
}
before(message $ request): execution(message drop$(message)) && args(*request) {
    removeNamespace(request);
}
before(message $ request): execution(message register$(message)) && args(*request) {
    removeNamespace(request);
}
before(message $ request): execution(message unregister$(message)) && args(*request) {
    removeNamespace(request);
}
before(message $ request): execution(message $associate$(message)) && args(*request) {
    removeNamespace(request);
}
before(message $ request): execution(message getMetadata$(message)) && args(*request) {
    removeNamespace(request);
}
```
The above code snippet shows the before advices in the **Remove Namespace** aspect. They are before advices so that the namespace information of the request message can be removed before the INFOD core system handles the request. They are woven into the execution of the public APIs (i.e., *CreatePublisher*, *RemoveConsumer*, etc) described in the specification [1]. If the aspect cannot find any namespace information in the message, it does nothing. Otherwise, it removes the namespace information from the message before letting the INFOD core system to handle the request.

### 4.11 Client Socket Aspect

This aspect is done for the non-functional requirement of the INFOD system “Remote Access to INFOD System”. For details of this requirement, please refer to Section 3.5.1. The main purpose of this aspect is to drop a request message to the C socket channel so that it can be picked up later by the INFOD system on the server side. By applying the criteria described in [4], it is a valid aspect. First, it is implemented for a crosscutting concern. All test cases created in this project originally access the INFOD system by direct procedural calls. Therefore, accessing the public APIs of the INFOD system becomes crosscutting. Second, implementing it as an aspect gives testers the flexibility to change or upgrade the remote access mechanism implemented in this project (e.g., C socket, web server, etc). Third, the remote access requirement is optional. It is a non-functional requirement. Furthermore, the test cases can be run without this aspect as long as there is an INFOD system deployed on the same local machine. Since it satisfies all the criteria described in [4], the remote access requirement on the client side can be implemented as an aspect. This aspect is a generic aspect because it does not have any
dependencies on the INFOD system. Developers can reuse this aspect on other systems without a lot of modifications.

```c
message around(message request): accessINFORegistryPointcut
{
    int sockfd, portno, n;
    struct sockaddr_in serv_addr;
    struct hostent *server;
    message response = NULL;

    portno = PORT_NO;
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        error("ERROR opening socket");
    server = gethostbyname(HOST);
    if (server == NULL){
        fprintf(stderr, "ERROR: no such host\n");
        exit(0);
    }
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    bcopy((char *) server->sin_addr,
          &serv_addr.sin_addr.s_addr,
          sizeof(serv_addr.s_addr);
    serv_addr.sin_port = htons(portno);
    if (connect(sockfd, &serv_addr, sizeof(serv_addr)) < 0)
        error("ERROR connecting");
    n = write(sockfd, request, strlen(request));
    if (n < 0)
        error("ERROR writing to socket");
    response = (message) malloc(sizeof(message) * BUFFER_SIZE);
    bzero(response, BUFFER_SIZE);
    n = read(sockfd, response, BUFFER_SIZE-1);
    if (n < 0)
        error("ERROR reading from socket");
    return response;
}
```

The above code snippet shows the *around* advice and the *pointcut* in **Client Socket** aspect. By examining the *pointcut* in the aspect, it intercepts the calls of all the public API specified in [1]. It is an *around* advice so that it can intercept the original procedural call and drop the original message to the socket. Then this aspect waits until there is a response message coming back from the socket. Finally, the aspect returns the response message to the intercepted modules.

### 4.12 High Performance Aspect

This aspect is done for the non-functional requirement of the INFOD system “High Performance”. For details of this requirement, please refer to **Section 3.5.2**. The main purpose of this aspect is to spawn a new process to handle an incoming request and allow
the INFOD managers to access the shared INFOD registry. To make sure no other
manager can access the registry, this aspect applies a lock to the registry when a manager
is already accessing it. Once the manager is done, the lock is released and the next
manager in the waiting queue is granted the permission to access the shared registry.

```
xmloctxt around(): execution(xmloctxt getRegistry())
{  // get the xml doc from shared memory.
    return xmlParseDoc(shmptr);
}
```

The code snippet above is an *around* advice for accessing the shared registry. The
*pointcut* is the `getRegistry()` internal method in the INFOD registry manager. It overrides
the original method so that it always returns the shared registry managed in this aspect.

```
char * around(char * request): call(char * handleRequest(char *)) & args(request)
{  ...
    if (pfid = fork()) < 0)
    {  printf("fork error\n");
        exit(-1);
    }  else if (pfid == 0)  
    {  close(Fd[1]);
        childRequest = (char *) malloc(sizeof(char) * 1000);
        if ((n = read(Fd[0], childRequest, 1000)) < 0)
        {  printf("read error from pipe\n");
            exit(-1);
        }  childResponse = proceed();
        close(Fd[2]);
        write(Fd[2], childResponse, 1000);
    }  else  
    {  close(Fd[0]);
        n = strlen(request);
        if (write(Fd[1], request, n) != n)
        {  printf("write error to pipe\n");
            exit(-1);
        }  if ((pfid = waitpid(pfid, &status, 0)) < 0)
        {  printf("waitpid error\n");
            exit(-1);
        }  close(Fd[2]);
        response = (char *) malloc(sizeof(char) * 1000);
        if ((n = read(Fd[0], response, 1000)) < 0)
        {  printf("read error from pipe\n");
            exit(-1);
        }  return response;
    }
```

The code snippet above is part of an *around* advice used to spawn a new child process to
handle an incoming request. It intercepts the `handleRequest()` method in the main
method of the INFOD registry system. This method determines the type of request and
then delegates the request to the right manager to handle. This advice then waits until
there is a response message coming back from the INFOD manager. Finally the advice returns the response message to the intercepted method.

```c
const char * around(): call($ insertRegistry(...)) || call($ updateRegistries(...)) || call($ dropRegistries(...)) || call($ insertRegistry(...)) || call($ searchRegistry(...)) || call($ getMetadata(...))
{
    char * result;
    struct semaphore * semoparg;

    int sem_id = getSemaphoreId();
    semoparg.sem_num = 0;
    semoparg.sem_op = -1;
    semoparg.sem_flg = SEM_UNDO;
    if ( semop(sem_id, semoparg, 1) == -1 )
        printf("failed to decrement semaphore\n");
    exit(-1);
}
result = proceed();

semoparg.sem_num = 0;
semoparg.sem_op = 1;
semoparg.sem_flg = SEM_UNDO;
if ( semop(sem_id, semoparg, 1) == -1 )
    printf("failed to increment semaphore\n");
exit(-1);

return result;
```

The code snippet above is an *around* advice that applies a semaphore to the shared registry. When a manager is about to access the shared registry, the shared registry will be locked by the semaphore. If another manager wants to access the shared registry at the same time, the semaphore puts that manager to sleep mode until the current manager releases the registry to prevent a race condition happen.

This aspect is a domain specific aspect. It is composed of two parts; one of them is to spawn a new child process to handle an incoming request, and the other one is to control the access to the INFOD registry. Since this aspect contains code to specifically handle the INFOD registry, it is less likely reusable in other systems.
5 System Product Family

A system product family contains a group of different versions of a software system. A software system can generate a number of versions by configuring it differently. Since the aspects implemented for the INFOD system are independent from each other, the aspects can be woven into the INFOD core system in any combination. Each combination becomes one configuration of the system or a member of the INFOD system product family.

Figure 24: Feature diagram of the INFOD system.

The feature diagram above shows the different configurations that can be generated for the INFOD system. The left side of the diagram lists the mandatory services provided by
the INFOD system. These mandatory services form the core of the system. The right
side of the diagram lists the optional features that can be activated in the INFOD system
and these are the aspects implemented in the system. After doing all the calculations,
there are 1024 different configurations of the INFOD system. The calculations are done
using the following combination formula.

\[
\binom{n}{r} \text{ or } nC_r = \frac{n!}{r!(n-r)!} = \frac{n(n-1)(n-2)\cdots(n-r+1)}{r!}
\]

where \( n \) is the total number of aspects available and \( r \) is the number of aspects to pick.

| 10 aspects choose 0 \( (10C_0) \) | 1 |
| 10 aspects choose 1 \( (10C_1) \) | 10 |
| 10 aspects choose 2 \( (10C_2) \) | 45 |
| 10 aspects choose 3 \( (10C_3) \) | 120 |
| 10 aspects choose 4 \( (10C_4) \) | 210 |
| 10 aspects choose 5 \( (10C_5) \) | 252 |
| 10 aspects choose 6 \( (10C_6) \) | 210 |
| 10 aspects choose 7 \( (10C_7) \) | 120 |
| 10 aspects choose 8 \( (10C_8) \) | 45 |
| 10 aspects choose 9 \( (10C_9) \) | 10 |
| 10 aspects choose 10 \( (10C_{10}) \) | 1 |
| **Total** | **1024** |

Table 9: The calculation of the number of INFOD system configurations.

<table>
<thead>
<tr>
<th>Family Member</th>
<th>Core Modules</th>
<th>Aspect Modules</th>
</tr>
</thead>
</table>
| INFOD base system with no aspects | Publisher Service  
Consumer Service  
Subscriber Service  
Subscription Service  
Vocabulary Service  
Registry Service  
Association Service | None |
| INFOD base system with 2 aspects | Publisher Service  
Consumer Service  
Subscriber Service  
Subscription Service  
Vocabulary Service  
Registry Service  
Association Service | Authorization  
Detach SOAP Header |
<table>
<thead>
<tr>
<th>INFOD base system with 9 aspects</th>
<th>Publisher Service</th>
<th>Authorization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consumer Service</td>
<td>Remove SOAP Header</td>
<td></td>
</tr>
<tr>
<td>Subscriber Service</td>
<td>Error Handling</td>
<td></td>
</tr>
<tr>
<td>Subscription Service</td>
<td>Message Filtering</td>
<td></td>
</tr>
<tr>
<td>Vocabulary Service</td>
<td>Logging and Tracing</td>
<td></td>
</tr>
<tr>
<td>Registry Service</td>
<td>Remove Referencing Elements</td>
<td></td>
</tr>
<tr>
<td>Association Service</td>
<td>Disable New Elements</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Remove Unused Elements</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Remove Namespace</td>
<td></td>
</tr>
</tbody>
</table>

Table 10: Examples of the INFOD system family members.

This project uses the weave adapters\(^1\) that can be downloaded from the [Aspect-oriented C](https://example.com) website to build each configuration of the INFOD system. There are a few benefits of using the weave adapters to configure the INFOD system. First, they help to reduce the footprints of the Makefile. Using the normal gcc\(^2\), the aspect files need to be compiled separately before being woven into the system. Using the weave adapters, the aspect files are compiled on the fly. Second, the C core files no longer require to have an extension of *.mc. They can be named with the conventional extension of *.c. The weave adapters automatically treat them as core files.

To use the weave adapters, a few environment variables need to be set or updated:

- **ACCHOME** – this is the root directory of the Aspect-oriented C utilities.
- **SRCROOT** – this is the root directory that contains the source files of the INFOD system.
- **PATH** – the path needs to be updated to also include the path of the `tacc`\(^3\) command and the `acc`\(^4\) command.

---

1. Weave adapters are a set of utilities that are used to weave aspects into large C projects.
2. The gcc utility is the C compile to compile C files.
3. `TACC` is the command to call the weave adapters to compile the core and aspect files.
4. `ACC` is the Aspect-oriented C command used to weave the aspects into the core system.
The weave adapters require one Makefile\(^1\) for each configuration. Since there are 1024 different configurations for this INFOD system, it becomes almost impossible to write a Makefile for each configuration by hand. Therefore, a Perl script is written to create these Makefiles. The Perl script creates the Makefiles using a template and adds the different aspect combination to each Makefile. Please refer to Appendix A.3 for an example of Makefile generated by the Perl script.

The generated Makefile describes how to configure a particular family member. First, it runs `tacc` to compile all the INFOD core and aspect files. The number of aspects being woven into the core varies in different family member. The command below shows a simple example of using the `tacc` command to weave aspects into core files to produce object files:

```
tacc -c -w publisherManager.c consumerManager.c authorization.acc detachSoap.acc
```

After the compilation is done successfully, it runs `accar` to build a library named `libinfod.a` using the object files produced by the compilation. After the library is built successfully, it can be linked to the C file containing the `main()` method (e.g., `server.c` in the remote access scenario) to produce the executable of the INFOD system. The command below shows a simple example of using the `accar` and `tacc` commands to build a library and link it to produce an executable.

```
accar rc libinfod.a publisherManager.o consumerManager.o
ranlib libinfod.a
```

```
tacc -o server -w server.o -L -linfod
```

---

\(^1\) A Makefile is used with the Unix make utility to determine which portions of a program to compile. It is a script that guides the make utility to choose the appropriate program files that are to be compiled and linked together.
6 Test Strategy

6.1 Approach

With over 1000 configurations, it is almost impossible to manually test the INFOD system. Since each configuration is treated as a new version of the INFOD system, if the system is being tested in the conventional way, it needs more than 1000 buckets of test cases to test the features offered by the core and the aspects; so for example, if each bucket has 10 test cases, the total number of test cases needed to thoroughly test the system will be more than 10000. Since the test cases are independent from each other, they need to have their own code to verify the test results. As more aspects are implemented in the system, the number of test cases grows dramatically and they soon become unmanageable.

To simplify the testing, the aspect-oriented approach can be applied to the verification testing of this project as well. There are seven managers in the INFOD core system. Since these managers are independent of each other, they can be tested individually. As a result, there is a test case developed for each manager to verify its functionalities. Since these test cases do not know any aspect in the system, they become the core test cases in the verification tests. For each aspect implemented for the system, there is an equivalent test case aspect implemented for the verification tests. When an aspect is woven into the INFOD core system, the associated test case aspect is brought into the core test cases as well. Testing the INFOD system this way does not need to define an explicit set of test
cases for each configuration. Depending on which test case aspects are woven into the core test cases, there is a set of test cases generated specifically for testing a particular configuration. The main purpose of these test case aspects is to verify the functionality provided by their associated INFOD system aspect. They do not interact with other test case aspects in the tests. These aspects are further explained in the next section.

The following table summarizes the differences between the two test approaches: aspect-oriented programming test approach and non-aspect-oriented programming test approach.

<table>
<thead>
<tr>
<th>Test Strategy using AOP approach</th>
<th>Test Strategy using non-AOP approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Requires fewer test cases to cover all the implemented features.</td>
<td>• Requires more test cases to cover all the implemented features.</td>
</tr>
<tr>
<td>• Requires more time to compile the test cases because the test case aspects need to be woven into the core test cases.</td>
<td>• Requires less time to compile the test cases because the system can be built into a library.</td>
</tr>
<tr>
<td>• Requires more thoughts when designing the test case aspects to make sure they do not contradict to each other.</td>
<td>• All test cases are independent from each other.</td>
</tr>
<tr>
<td>• Test data sets are managed in the core test cases and the test case aspects. The test case aspects generate test data sets from those in the core test cases.</td>
<td>• Test data sets are finite and managed solely by each individual test case.</td>
</tr>
</tbody>
</table>

Table 11: The differences between AOP-based and non-AOP based test approaches.

Since it is nearly impossible to run the tests for each configuration of the INFOD system by hand within a reasonable time, a test harness is developed in this project to automate the verification testing of the system. The Perl script mentioned in the previous chapter is used to setup part of the test environment for the INFOD system. Recall that the main purpose of this Perl script is to create a Makefile for each configuration of the INFOD
system. The Perl script also creates a directory for each configuration and then puts the Makefile in the directory. After the test environment is ready, the test harness can be launched to run the tests. The test harness does the following:

1. Change the working directory to a test folder.
2. Run the Makefile in the test folder. This will build and run the test cases in the test folder.
3. Collect the test results.
4. Go to the next test folder and repeat step 2 and 3.

6.2 Verification Aspects

These verification aspects verify the functionality provided by the INFOD system aspects. The verification is done by weaving these aspects to the core test cases.

6.2.1 Attach SOAP Header Verification Aspect

This aspect is activated when the Attach SOAP Header aspect is woven into the INFOD system. For more details of the Attach SOAP Header aspect, please refer to Section 3.3.1 and Section 4.1. The main purpose of this aspect is to verify if a SOAP header is attached to the response message generated for an incoming request.
The above code snippet shows the *around* advice of this verification aspect. The `compare(message, message)` is a function in the core test cases to verify the actual response with the expected response. This aspect verifies if there is a SOAP header presented in the actual response. If the response does not contain a SOAP header, it terminates the test with a return code of -1. Otherwise, since the core test cases do not expect a SOAP header, this aspect removes the SOAP header before proceeding the call.

### 6.2.2 Remove SOAP Header Verification Aspect

This aspect is activated when the *Remove SOAP Header* aspect is woven into the INFOD system. For more details of the *Remove SOAP Header* aspect, please refer to Section 3.3.2 and Section 4.2. The main purpose of this aspect is to verify if a SOAP header is already removed from the request message before it is handled by INFOD system.
The above code snippet shows the *around* advice of this verification aspect. It intercepts the execution of all public APIs specified in [1]. Before proceeding the API call, this aspect verifies if there is a SOAP header in the request message. If it contains a SOAP header, it terminates the test with a return code of -1.

### 6.2.3 Message Filtering Verification Aspect

This aspect is activated when the Message Filtering aspect is woven into the INFOD system. For more details of the Message Filtering aspect, please refer to Section 3.3.3 and Section 4.3. The main purpose of this aspect is to verify if the property constraints specified in the INFOD elements are handled by the Message Filtering aspect.
The above code snippet shows the *around* advice of this verification aspect. It intercepts the main function of the core test case for INFOD subscriptions. Without the **Message Filtering** aspect, all the notification messages generated by the **Subscription Manager** are stored in a message queue. Otherwise, only certain notification messages are generated and stored in the queue. If it cannot find all the expected messages in the queue, it terminates the test with a return code of -1.

### 6.2.4 Drop Unused INFOD Elements Verification Aspect

This aspect is activated when the **Drop Unused INFOD Elements** aspect is woven into the INFOD system. For more details of the **Drop Unused INFOD Elements** aspect, please refer to Section 3.3.4 and Section 4.4. The main purpose of this aspect is to verify if an INFOD element is removed only if it is no longer referenced by other INFOD elements.

```c
message around(message request); { call(message dropSubscriber(message)) ||
          call(message dropSubscription(message))) &&
          arg1(request)
          {
            response = proceed();
            target = searchRegistry(xpathExpr);
            size -= (target->nodes->eval) / target->nodes->eval->nodeVal : 0;
            if (size == 0)
              {
                if (strcmp(this->targetName, "dropSubscriber") == 0)
                  {
                    // Check to see if the subscriber is still referenced.
                    target = searchRegistry("/SubscribersReference");
                    size -= (target->nodes->eval) / target->nodes->eval->nodeVal : 0;
                    for (i=0; i<size; i++)
                      {
                        refNode = target->nodes->eval->nodeTab[i];
                        refValue = xmlNodesGetContent(refNode);  
                        if (strcmp(refValue, exprValue) == 0)
                          {
                            printf("TEST FAILED: unitest ifunused: the subscriber is removed\n");
                            exit(-1);
                          }
                      }
                }
                else
                  {
                    if (strcmp(this->targetName, "dropSubscription") == 0)
                      {
                        printf("TEST FAILED: unitest ifunused: the subscription is not removed\n");
                        exit(-1);
                      }
                  }
            }
            return response;
          }
```

The above code snippet shows the *around* advice of this verification aspect. This verification aspect changes the execution mode of two particular requests to "IF
After these two requests are handled, this aspect verifies if the corresponding INFOD element is removed only if no other INFOD elements reference it. If the verification fails, it terminates the test with a return code of -1.

### 6.2.5 Disable New INFOD Elements Verification Aspect

This aspect is activated when the **Disable New INFOD Elements** aspect is woven into the INFOD system. For more details of the **Disable New INFOD Elements** aspect, please refer to **Section 3.3.5** and **Section 4.5**. The main purpose of this aspect is to verify if an INFOD element can no longer be referenced by other INFOD elements.

```c
message around(message request): {
    call(message_dropPublisher(message));
    call(message_disassociateVocabulary(message));
    args(request) && infunc(main)
    {
      ...
      response = proceed();
      root = parse(request);
      ...
      if (eprValue != NULL)
      {
        size = strlen(xpathExpr) + strlen(eprValue) + strlen("[Epr='']") + 10;
        xpathExpr2 = (char *) malloc(size * sizeof(char) * size);
        sprintf(xpathExpr2, "%s[Epr='%s']", xpathExpr, eprValue);
      }
      target = searchRegistry(xpathExpr2);
      size = (target->nodesetval) ? target->nodesetval->nodesize : 0;
      if (size > 0)
      {
        entitynode = target->nodesetval->nodeset[0];
        if (xGetProp(entitynode, "disable") == NULL)
        {
          return response;
        }
        else
        {
          printf("TEST FAILED: unittest_disableNew; The INFOD element is not disabled.\n");
          exit(-1);
        }
      }
      return response;
    }
}
```

The above code snippet shows the **around** advice of this verification aspect. This verification aspect changes the execution mode of two particular requests to `DISABLE NEW`. After these two requests are handled, this aspect verifies if the corresponding INFOD element is disabled. This is done by checking if the disable flag of the INFO element is set. If the verification fails, it terminates the test with a return code of -1.
6.2.6 Authorization Failure Verification Aspect

This aspect is activated when the Authorization Failure aspect is woven into the INFOD system. For more details of the Authorization Failure aspect, please refer to Section 3.3.7 and Section 4.6. The main purpose of this aspect is to verify if the API handling the request without authorization returns an exception message.

```java
message around(): call(message createSubscription(message)) ||
    call(message dropSubscription(message)) && infunc(main)
{
    ...  
    size = 0;  
    response = proceed();  
    root = parse(response);  
    if (stricmp(this->targetName, "createSubscription") == 0)  
    {  
        target = searchboc(root, //Infod:CreateEntityAuthorizationFailure);  
        size = (target->nodeSetVal) ? target->nodeSetVal->nodeVal : 0;  
    }  
    else if (stricmp(this->targetName, "dropSubscription") == 0)  
    {  
        target = searchboc(root, //Infod:DropEntityAuthorizationFailure);  
        size = (target->nodeSetVal) ? target->nodeSetVal->nodeVal : 0;  
    }  
    if (size > 0)  
    {  
        ...  
    }  
    else  
    {  
        return "TEST FAILED! The user should not be authorized!";  
    }
}
```

The above code snippet shows the around advice of this verification aspect. This verification aspect changes the user who issues the requests. After these requests are handled, this aspect verifies if the return response is a fault message. If the response is not a fault message, it replaces the response with an error message which will eventually fail the test when the core test case verifies the response.

6.2.7 Error Handling Verification Aspect

This aspect is activated when the Error Handling aspect is woven into the INFOD system. For more details of the Error Handling aspect, please refer to Section 3.3.8 and
Section 4.7. The main purpose of this aspect is to verify if the API handling the request with errors returns an exception message.

```c
static int compare(message result, message expected)
{
    int status = -1;
    status = strcmp(result, expected, strlen(expected));
    if (status != 0)
    {
        printf("result -->\n");
        printf("%s\n", result);
        printf("expected -->\n");
        printf("%s\n", expected);
        exit(-1);
    }
    return status;
}

after(message response): call(message createSubscription(message)) & func(verifyException) & result(response)
{
    message expected;
    expected = "<?xml version="1.0"?>
    <infod:MissingRequiredParameterFault xmlns:infod="http://www.ggf.com/infod"
    compare(response, expected);
}

after(message response): call(message associateVocabulary(message)) & func(verifyException) & result(response)
{
    message expected;
    expected = "<?xml version="1.0"?>
    compare(response, expected);
}
```

The above code snippet shows the after advices of this verification aspect. This verification aspect contains the expected fault messages for the requests with errors. After the requests are handled, this aspect intercepts and verifies if the return response matches the expected fault messages. If it does not, it terminates the test and returns a status of -1.

### 6.2.8 Remove INFOD References Verification Aspect

This aspect is activated when the **Remove INFOD References** aspect is woven into the INFOD system. For more details of the **Remove INFOD References** aspect, please refer to [Section 3.3.9](#) and [Section 4.9](#). The main purpose of this aspect is to verify if the INFOD elements referencing the removed element are also removed.
The above code snippet shows the *around* advice of this verification aspect. This verification aspect tries to find the INFOD elements that reference the removed element. If these elements are found in the registry, it terminates the test and returns a status code of -1.

### 6.2.9 Remove Namespace Verification Aspect

This aspect is activated when the **Remove Namespace** aspect is woven into the INFOD system. For more details of the **Remove Namespace** aspect, please refer to Section 3.3.11 and Section 4.10. Unlike other verification aspects, this aspect does not verify anything. Because the request messages in the core test cases do not have any namespace information, this aspect is used to add namespace information to the request messages. The verification is handled by the core test cases because they always expect the messages to have no namespace information. Therefore, if the core test cases do not complain, it implies that the **Remove Namespace** aspect works properly.
The above code snippet simply adds namespace information to the request messages before they are handled by the INFOD core system.
7 Evaluation

This chapter summaries the observations and insights realized during the development of the INFOD system.

7.1 Lessons Learned

A few things are learned from the development of this project, which are different from the conventional way of developing software systems.

First, the core and aspects should not know the existence of each other, i.e., the core does not know about the aspects. Doing it this way makes sure that the core of the INFOD system can work properly without the aspects. Furthermore, it makes the code simpler and easier to maintain since the code has fewer dependencies. This is similar to the goal of the object-oriented approach in which developers try to decouple the objects in the system as much as possible.

Second, the implementation should be divided into two stages. The first stage is the implementation of the core of the system. In this stage, developers should concentrate on implementing and testing on the core of the system. They should make sure that the core is fully functional without any problems. In the next stage, developers should then concentrate on the implementation and the testing of the aspects. Implementing the system this way helps debugging the problems more easily. If the core is fully tested, i.e.,
error-free and a problem is found when an aspect is woven into the system, developers can easily isolate the problem because the problem is very likely related to the aspect.

These lessons learned can serve as some additional guidelines when developing another project using the aspect-oriented programming model.

### 7.2 Implementation Evaluation

The implementation of this project uses the aspect-oriented programming approach. The tables below summarize some quantitative results measured in this project. All the evaluations are done in CygWin in Windows XP SP2 on an Intel Pentium M 1600MHz processor, 2GB of RAM machine.

<table>
<thead>
<tr>
<th>Number of System Aspects</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Verification Aspects</td>
<td>9</td>
</tr>
</tbody>
</table>

**Table 12: Number of aspects in the INFOD system.**

There are fewer verification aspects because some of the system aspects cannot be verified. They are the **Logging and Tracing** aspect, the **Remote Access** aspect and the **High Performance** aspect.

<table>
<thead>
<tr>
<th></th>
<th>Number of Lines of Code (LOC) with Comments</th>
<th>Percentage (%) of Total Number of Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>2512</td>
<td>46.4%</td>
</tr>
<tr>
<td>System Aspects</td>
<td>2905</td>
<td>53.6%</td>
</tr>
<tr>
<td>Attach SOAP Header</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>Authorization</td>
<td>299</td>
<td></td>
</tr>
<tr>
<td>Remote Access</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>Detach SOAP Header</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>Disable New INFOD Element</td>
<td>340</td>
<td></td>
</tr>
<tr>
<td>Error Handling</td>
<td>722</td>
<td></td>
</tr>
<tr>
<td>Message Filtering</td>
<td>538</td>
<td></td>
</tr>
<tr>
<td>Logging and Tracing</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>--------------------------</td>
<td>----</td>
<td></td>
</tr>
<tr>
<td>Remove Namespace</td>
<td>72</td>
<td></td>
</tr>
<tr>
<td>High Performance</td>
<td>203</td>
<td></td>
</tr>
<tr>
<td>Remove INFOD References</td>
<td>217</td>
<td></td>
</tr>
<tr>
<td>Remove INFOD Unused Element</td>
<td>192</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>5417</strong></td>
<td></td>
</tr>
</tbody>
</table>

Table 13: Number of lines of code in the INFOD system.

<table>
<thead>
<tr>
<th>Number of INFOD system configurations</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of lines of code in one configuration (the core with 10 aspects)</td>
<td>5139</td>
</tr>
<tr>
<td>Time to build one configuration (the core with 10 aspects)</td>
<td>114 sec</td>
</tr>
<tr>
<td>Time to build all configurations</td>
<td>32 h 41 min</td>
</tr>
<tr>
<td>Number of tests per configuration</td>
<td>9</td>
</tr>
<tr>
<td>Time to test one configuration (the core with 10 aspects)</td>
<td>22 min</td>
</tr>
<tr>
<td>Time to test all configuration</td>
<td>67 h 20 min</td>
</tr>
</tbody>
</table>

Table 14: Time required to build and test the INFOD system.

By observing the time needed to build a configuration, as there are fewer aspects woven into the core, the system is built faster.

<table>
<thead>
<tr>
<th>Total number of configurations run against their use cases</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of configurations run against their test cases passed</td>
<td>1024</td>
</tr>
<tr>
<td>Number of configurations run against their test cases failed</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 15: Test case results of INFOD system.

### 7.3 Performance Evaluation

The purpose of this evaluation is to evaluate the overhead introduced by the aspects. A small benchmark has been defined to evaluate the performance against two particular configurations. One configuration is composed of the core and four selected aspects (i.e., Attach SOAP Header aspect, Detach SOAP Header aspect, Remove Namespace aspect and Message Filtering aspect). The second configuration is similar to the first one. But
instead of implementing those four optional features as aspects, they are implemented in C as default features built into the INFOD system. The evaluation is done by running against two large data sets. One data set contains 500 requests and the other one contains 1000 requests. An example of the test data is attached in the Appendix A.4. The following tables are the performance metrics gathered in this evaluation. Both tables show a small overhead introduced by the aspects. This evaluation is done in CygWin in Windows XP SP2 on an Intel Pentium M 1600MHz processor, 2GB of RAM machine.

<table>
<thead>
<tr>
<th>Run</th>
<th>Time Required to Handle 500 Requests (in seconds)</th>
<th>System with Features Built-in</th>
<th>System with Aspects</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>29</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>29</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>29</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>29</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>29</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>29</strong></td>
<td><strong>29.6</strong></td>
<td></td>
</tr>
</tbody>
</table>

Table 16: Performance metrics on handling 500 requests.

<table>
<thead>
<tr>
<th>Run</th>
<th>Time Required to Handle 1000 Requests (in seconds)</th>
<th>System with Features Built-in</th>
<th>System with Aspects</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>99</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>99</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>99</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>99.4</strong></td>
<td><strong>100</strong></td>
<td></td>
</tr>
</tbody>
</table>

Table 17: Performance metrics on handling 1000 requests.

<table>
<thead>
<tr>
<th>Run</th>
<th>Time Required to Handle 2000 Requests (in seconds)</th>
<th>System with Features Built-in</th>
<th>System with Aspects</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>368</td>
<td>370</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>368</td>
<td>369</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>369</td>
<td>370</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>369</td>
<td>369</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>369</td>
<td>369</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Run</th>
<th>Time Required to Handle 2000 Requests (in seconds)</th>
<th>System with Features Built-in</th>
<th>System with Aspects</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>368</td>
<td>370</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>368</td>
<td>369</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>369</td>
<td>370</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>369</td>
<td>369</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>369</td>
<td>369</td>
<td></td>
</tr>
</tbody>
</table>
### 7.4 Evaluation on the Approaches to Identify Aspects

Two approaches have been used to identify the aspects in this project. One approach is described in [3] while the other one is described in [4]. The approach described in [3] is simpler to use because the aspects are easier to visualize from the use case diagrams. On the other hand, the approach described in [4] involves more thoughts and there is no diagram or graph to help to identify the aspects. In general, both approaches are very similar because most of the aspects identified in this project can be identified by both approaches. However, there are still some differences between the two approaches. One of the main differences between the two is the criteria derived from [4]: the feature can be easily upgradeable without affecting the rest of the system. In the criteria described in [3], aspects are identified only from a list of features in the alternate flow or sub-flow of the use cases. If the features are not in these flows, they are not eligible to become an aspect. However, if using the criteria described in [4], some of the features in the main flow can also be implemented as aspects if they will likely be upgraded in the future. The socket client aspect is a good example to illustrate this difference between the two approaches. The main purpose of this aspect is to drop the request to the socket channel and wait for the INFOD system to handle the request on the server side. If using the approach described in [3], this action will be in the main flow of the use case because it is mandatory in a distributed environment. However, if using the approach described in [4], it should be implemented as an aspect because it offers a few benefits. Besides sockets, there are other protocols that can be used in a distributed environment. If sockets are not

<table>
<thead>
<tr>
<th>Average</th>
<th>368.6</th>
<th>369.4</th>
</tr>
</thead>
</table>

Table 18: Performance metrics on handling 2000 requests.
supported or there is a better protocol already implemented in a particular environment, it will be a good idea to switch the INFOD system to use that protocol instead. Since the socket code is maintained in one common place, it becomes very easy to switch to a different protocol. Because of this reason, the socket client has been implemented as an aspect in this project. As a conclusion, using either approach is good enough to identify aspects but using both approaches may help to identify more aspects that may not be seen if using just one approach.

7.5 Missing Requirements in INFOD Base Specification

This section summarizes the requirements missing in the INFOD Base Specification [1].

7.5.1 Manipulating Data Sources

The INFOD specification [1] is the building ground of this project. The public functions defined in the core system are the APIs defined in the base specification. Recall that the INFOD system is a system that uses a notification mechanism to make data sources available to consumers. However, the base specification does not specify on how data sources are available in the registry or where the data sources are located (e.g., a database that can be reached by the INFOD system). Therefore, a few assumptions are made when designing the system to make the implementation easier.

1. The data interested to the consumers are stored in the INFOD registry.

2. There are APIs that are implemented by third parties available in the INFOD system. These APIs allow external services to insert and manipulate the data in the INFOD registry.
3. In unit testing, it is assumed that there are some data sources available in the INFOD registry for testing purposes.

### 7.5.2 Authorization Requirement

According to the base specification [1], the public APIs can return a fault message describing the authorization failure to access the INFOD system. However, it does not specify how authorization is done in the INFOD system. As a result, a very simple authorization mechanism has been applied to the system so that the **Authorization Failure** aspect can still be implemented.

The authorization check is done in the SOAP header of the message. To make things simple, all authorized users who can access the INFOD system are stored in an XML file, `users.xml`. The authorization check determines if the username specified in the SOAP header is in the list of authorized users in the XML file. If the user is not authorized, the **Authorization Failure** aspect generates a fault message; otherwise, it allows the request to proceed. Below is how the `users.xml` looks like.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
  <user>johns</user>
  <user>flannan</user>
</users>
```

### 7.6 Aspect-Oriented Design

This section summarizes some insights and observations realized on aspect-oriented design while working on this project.
7.6.1 Logging and Tracing

Logging and tracing is a typical aspect to be implemented in a system. However, if a detailed logging is needed by a developer, just having an aspect to do the job may not be enough. If the developer wants to have some logging in the middle of the function, the aspect cannot do it because an aspect can be invoked only before and after the execution or call of a function being monitored. The workaround is to break down the system in very small functions, but this is probably not practical to the developers. There are two main purposes why developers may want to use functions. One is for sharing the code with other modules; the other one is to break down a large piece of code into smaller pieces for easier maintenance and make it more readable. Since developers tend to group related code in the same function, a function may still be couple hundred lines long. If developers want to provide some tracing capability among those lines of code, they have to do it outside of the logging aspect and thus all the logging and tracing may not be done in one central place.

7.6.2 Realizing Crosscutting Concerns

Like other programming approaches, developers identify possible aspects in both design and implementation phases. In some cases, additional crosscutting concern may only be identified in the implementation phase because it depends on how developers code their systems. For example, when designing the INFOD system, removing an INFOD entity from the registry and its referencing entities is considered as part of the core system because it is a fundamental requirement in the specification [1]. However, when implementing the core system, it turns out that the code for removing the referencing entities can be isolated and becomes an individual aspect. The reason is that it can be
considered as an optional feature and the core system does not break without it. Furthermore, it is crosscutting because there are a few APIs sharing this feature. As a result, there is an additional aspect implemented to remove the INFOD entities that references the removing entity. On the other hand, aspects identified in the design phase may later become a non-feasible aspect to implement in the implementation phase. The reason is that during the implementation, it may find out the aspect is not optional enough to be excluded from the core system. For example, when designing the INFOD system, there is an aspect for manipulating the INFOD registry. It is crosscutting and implementing it as an aspect allows developers to change or upgrade the registry infrastructure easily without impacting the core code. However, when implementing the system, it becomes obvious that the core system cannot live without it. Without it, the INFOD system cannot handle some of the requests because some managers need other INFOD entities from the registry before they can successfully process their requests. As a result, the manipulation of the INFOD registry is implemented as part of the core instead of an independent aspect.

### 7.7 Aspect-oriented C Language

This section summarizes some insights and observations realized on the Aspect-oriented C language.

#### 7.7.1 Aspects of Aspects

Aspect-oriented C is the aspect-oriented programming language that is used to implement the INFOD system. During the implementation phase of this project, it becomes more obvious that having the capability to weave aspects into other aspects in Aspect-oriented
C is very important. For example, the logging and tracing is implemented as an aspect in this project. If aspects of aspects are supported in Aspect-oriented C, the logging and tracing can be done more thoroughly in this project. Since half of the code is done in aspects, the logging and tracing aspect can only provide logging information for half of the project, i.e., the core. If the logging and tracing aspect can also be implemented as an aspect of aspect, it can help debugging significantly because it can provide debugging information everywhere in the code.

Another example is the **Attach SOAP Header** aspect which attaches a SOAP header to a message. If Aspect-oriented C supports aspects of aspects, this may reduce the number of join points in the core program. Currently, this aspect inserts a join point in each of the public API specified in [1]. Since there are a few utility methods in the core program that create response messages for these APIs, the join points can be inserted into those methods instead. The join points must be in the public APIs because there are other aspects that can intercept the execution of these APIs and they can return response messages to the caller instead. Without the support of aspects of aspects, the response messages generated by these aspects do not have a SOAP header attached to it.

Since the aspects of aspects feature is supported in the latest version of Aspect-oriented C, the **Attach SOAP Header** aspect has been modified to see how it impacts the original design of the aspect. The following are some observations found after implementing the aspect to use the aspects of aspects feature. First, it does reduce the number of join points inserted into the core, from 20 join points reduced to 8 join points. Second, the SOAP
header is now attached to the message right at where the message is created rather than attaching the SOAP header to the message when it is about to leave the INFOD system.

The code now makes more sense because the response message is considered to be complete upon leaving the utility method. The only disadvantage of this feature is that the aspect using the aspects of aspects feature may have dependencies on other aspects because it also weaves into other aspects. If a developer decides to implement a new aspect that generates response messages for the public APIs specified in [1], he will need to modify this aspect so that it knows about the new aspect and it can attach a SOAP header to the response messages created by the new aspect.

Below is the code snippet that shows the advices in the **Attach SOAP Header** aspect with aspects-of-aspects support. The `createFault(…)` and `createAuthorizationFailure(…)` in the first `around` advice are defined in the **Authorization Failure** aspect and the **Error Handling** aspect. With the `acc -a` option, this aspect is also woven in the **Authorization Failure** aspect and the **Error Handling** aspect to attach a SOAP header to the message generated by these two aspects.

```java
// This advice intercepts the API calls to wrap the returned message in a SOAP message.
message around(): call(message createResponseMessage(...)) || call(message createFault(...)) || call(message createAuthorizationFailure(...))
{
    xmlDocPtr bodyptr;
    xmlDocPtr bodyrootnode;
    xmlDocPtr soapdoc;
    wchar_t * result;
    int size;
    message response;
    response = proceed();
    bodyptr = parse(response);
    bodyrootnode = xmlDocGetRootElement(bodyptr);
    soapdoc = createSoapMessage(kul, bodyrootnode);
    xmlDocDumpMemory(soapdoc, &result, &size);
    return result;
}
```
7.7.2 Accessing Local Variables

Currently, the aspects in Aspect-oriented C can only access the arguments and the return value of the intercepted core methods. It will be useful if the aspects can also access the local variables declared in the intercepted method. Without this capability, some of the crosscutting concerns may not be addressed by aspects. For example, destroying a structure pointer can be implemented as an aspect. In the INFOD system, an XML document pointer has been used everywhere to point to the XML message structure. This pointer is defined in many methods in the INFOD system. If aspects can access local variables in the intercepted method, destroying the XML document pointer can be implemented as an aspect and this makes sure that the memory referenced by these pointers is never forgotten to free up properly. So, for example, one can have something like this:

```c
after(xmlDocPtr doc): execution($ insertRegistry(...) && localVar(doc)
{
  if (doc != NULL)
  {
    xmlFreeDoc(doc);
  }
}
```

The above code snippet is an after() advice. The `localVar` is a new keyword to retrieve the local variables from the intercepted functions. The name and type of the variable defined in the advice must match those defined in the intercepted functions. The main
purpose of this advice is to free the memory allocated for the parsed XML object referenced by the pointer ‘doc’.

In addition, having the ability to access local variables can help manipulate the data interested to the aspects. In all public APIs specified in [1], the only argument to these APIs is a character string that represents an XML message. Since these core methods need to parse the message to handle the request, they already have local variables holding the parsed information such as the publisher ID, the customer ID, etc. Since aspects cannot access these local variables, they need to parse the message themselves to get these information and thus it can impact the performance of the system. The current workaround is to create a dummy method which has the parsed information as its arguments and then create a pointcut in the aspect to intercept this dummy method. However, this dummy method does nothing but increases the footprint of the core and thus it is not a good practice of programming.

7.7.3 Modifying Results in After() Advice

If there is a way to modify the returning value of an intercepted function in the after advice, it will make the implementation of some aspects much simpler. Because this cannot be done in the after advice, the around advice must be used to return a different value. In some of the aspects implemented for this system, they have to let the intercepted function to proceed and finish before they can do their work.

For example, in the Attach SOAP Header aspect, it must let the intercepted function to generate a returning message before it can attach a SOAP header to the message. Since
the *after* advice does not allow modifying the returning value, the aspect cannot use an *after* advice to attach a SOAP header to the message. Instead, it has to use an *around* advice.

```java
message around(): execution($ createSubscriptionNotification(...)) { 
    xmlDocPtr bodyDoc;
    xmlDocPtr soapDoc;
    xmlChar * result;
    int size;
    message response;
    response = proceed();
    bodyDoc = parse(response);
    bodyRootNode = xmlDocGetRootElement(bodyDoc);
    soapDoc = createsoapmessage("http://www.gef.org/infd/INFONotify/subscriptionNotification", bodyRootNode);
    xmlFreeMemory(soapDoc, &result, &size);
    return result;
}
```

By examining the above code snippet, the *proceed()* call is executed every time and thus the *around* advice does not seem to be appropriate to use here. The *around* advice should instead be used to provide an alternative to proceeding the intercepted function. The *around* advice is used here simply because of the lack of modifying returning value in *after* advices. If looking closely to the main purpose of the Attach SOAP Header aspect, it is to attach the header to the returning message and thus it sounds more correct to use an *after* advice to attach the header to the message.

### 7.8 Idioms

During the later stage of the implementation phase, *named pointcuts* have been used more frequently in the aspect codes because they make the code look simpler. The code snippet below is an example of a *named pointcut*.

```java
pointcut accessINFO00RegistryPointcut(message request): \call{create\$\(\text{message}\) &\&
    \call{replace\$\(\text{message}\) &\&
    \call{drop\$\(\text{message}\) &\&
    \call{register\$\(\text{message}\) &\&
    \call{unregister\$\(\text{message}\) &\&
    \call{get\$\text{MetaData\(\text{message}\) &\&
    \call{args\(\text{request}\) &
```
named pointcut becomes so long that makes the advice look messy. As a result, named pointcuts are used to address this issue. They are defined at the beginning of the aspect files. With a meaningful name for these pointcuts, it becomes easier to understand where the advice intercepts and makes the advice look cleaner.

For good coding practice, long functions are avoided in the aspects and the core. Long functions often give readers a difficult time of reading and understanding what the function does. The average number of lines of each function found in the aspects is about 50 lines. If a function is long, it is broken down into smaller functions to increase its readability.

Another common idiom used in coding the aspects and the core is the short form of if statement. Instead of having the regular format:

```c
int size = 0;
if (strcmp(a, “hello”) == 0)
  size = 1;
else
  size = 2;
```

the if statement is written as:

```c
int size = strcmp(a, “hello”) == 0 ? 1 : 2;
```

This helps to reduce the footprint of the project and thus this idiom has been widely used throughout the project.
8 Conclusions

Aspect-oriented programming is a new programming approach to help separating crosscutting concerns which cannot be easily separated in conventional programming models. Although there are many research projects exploring how aspect-oriented programming can impact the development of software applications, most of them are done in a refactoring fashion and most of them are done using AspectJ. To realize how aspect-oriented programming can impact the software development cycle as a whole and on procedural programming languages such as C, this project uses Aspect-oriented C to implement the Information Dissemination (INFOD) system, a system that uses a notification mechanism to make data sources available to consumers.

Throughout the software development, a number of aspects are identified using two approaches. One approach identifies aspects by analyzing the use case scenarios of the system. This approach has two main criteria which are described in [3]. The second approach identifies aspects by applying the criteria derived from the horizontal decomposition principles described in [4]. These two approaches are pretty similar because most aspects implemented in this project can be identified by either approach. However, there are still a few cases in which the criteria of one approach are different enough to identify some aspects that the other approach cannot be identified, e.g., the Client Socket aspect.
By analyzing the code, we successfully illustrate how bad crosscutting concerns can be coupled to a typical software system. By separating all crosscutting concerns identified in the INFOD system, the core part is only half the size of the whole INFOD system, i.e., 46% core vs. 54% aspects in number of lines of code.

When implementing the INFOD system, a few lessons are learned by using aspect-oriented programming in the software development. We realize that the aspects and the core should be totally separate; in other words, they should not know the existence of each other. It is because the aspects are not mandatory to the base system and thus the base system should still be functional without the aspects. Also, this makes the code cleaner and easier to understand. Furthermore, since the aspects are totally independent from the core system, it is a good idea to fully test the core system before implementing and testing the aspects. In this case, it helps debugging much easier because if a problem comes up after weaving an aspect into the system, we can assume that the problem is most likely caused by the weaving aspect. This can save a lot of time in locating the error.

There are a few areas that can be focussed in the future work. This project has implemented the INFOD system to be accessed in two ways. One is to access the system directly from procedural calls and the other way is to access the system remotely via C sockets. The INFOD system is originally designed to be deployed on a web environment to handle requests issued by external web services. Due to the size and the time constraints of this project, it does not have a chance to extend this system further to
deploy on a web server environment. Therefore, one of the areas that can be focussed in the future is to make the INFOD system deployable on a real client and server environment so that it can be used to handle requests issued by external web services.

In addition, another area of the INFOD system that can be explored in the future is security. Due to the time constraints of this project, security is not taken into consideration when implementing the INFOD system. However, security has become more and more important in today’s web applications. Therefore, security will become an issue once the INFOD system is ready to handle requests from real web services. The main concern of security to be addressed in the INFOD system is the security of exchanging messages between publishers and consumers. This can be a broad area of research which includes how and who creates security tokens and how message encryption can be done. Since the INFOD system is implemented using the aspect-oriented programming model, one should also consider if security can be implemented in an aspect.
Bibliography


Appendix A

A.1 Use Case Diagram

The following use cases provide an overview of the typical usages of the INFOD system.

![Use Case Diagram](image)

Figure 25: The use case describing how different actors create an INFOD entity in the INFOD system.

A summary of steps describing the use case above are as follows:

1. The use case begins with the arrival of a request message.
2. The SOAP header is removed from the message.
3. The message is passed into an XML parser.
4. The manager then validates the request message.
5. The manager inserts the new entity into the INFOD registry.
6. The manager creates a response message.
7. A SOAP header is added to the response message.
8. The manager returns the message to the requesting service.

9. The use case terminates.

A summary of steps describing the use case above are as follows:

1. The use case begins with the arrival of a request message.
2. The SOAP header is removed from the message.
3. The message is passed into an XML parser.
4. The manager validates the request message.
5. The manager updates the associated registry with the given EPR.
6. The manager creates a response message.
7. A SOAP header is added to the response message.
8. The manager returns the message to the requesting service.

Figure 26: The use case describing how different actors replace an INFOD entity in the INFOD system.
9. The use case terminates.

A summary of steps describing the use case above are as follows:

1. The use case begins with the arrival of a request message.
2. The SOAP header is removed from the message.
3. The message is passed into an XML parser.
4. The manager validates the request message.
5. The manager searches for a list of elements that references this entity.
6. The manager removes this entity and all elements referencing this entity from the INFOD registry.
7. The manager creates a response message.
8. A SOAP header is added to the response message.
9. The manager returns the message to the requesting service.
10. The use case terminates.

A summary of steps describing the use case above are as follows:

1. The use case begins with the arrival of a request message.
2. The SOAP header is removed from the message.
3. The message is passed into an XML parser.
4. The manager validates the request message.
5. The manager inserts the new vocabulary into the INFOD registry.
6. The manager creates a response message.
7. A SOAP header is added to the response message.
8. The manager returns the message to the requesting service.
9. The use case terminates.
A summary of steps describing the use case above are as follows:

1. The use case begins with the arrival of a request message.
2. The SOAP header is removed from the message.
3. The message is passed into an XML parser.
4. The manager validates the request message.
5. The manager searches for a list of elements that references this vocabulary.
6. The manager removes this entity and all elements referencing this entity from the INFOD registry.
7. The manager creates a response message.
8. A SOAP header is added to the response message.
9. The manager returns the message to the requesting service.

10. The use case terminates.

A summary of steps describing the use case above are as follows:

1. The use case begins with the arrival of a request message.

2. The SOAP header is removed from the message.

3. The message is passed into an XML parser.

4. The manager validates the request message.

5. The manager inserts the new property vocabulary instance into the INFOD registry.

6. The manager creates a response message.

7. A SOAP header is added to the response message.

8. The manager returns the message to the requesting service.

9. The use case terminates.
Figure 31: The use case describing how different actors remove a property vocabulary instance from the INFOD registry.

A summary of steps describing the use case above are as follows:

1. The use case begins with the arrival of a request message.
2. The SOAP header is removed from the message.
3. The message is passed into an XML parser.
4. The manager validates the request message.
5. The manager searches for a list of elements that references this vocabulary instance.
6. The manager removes this instance and all elements reference this instance from the INFOD registry.
7. The manager creates a response message.
8. A SOAP header is added to the response message.
9. The manager returns the message to the requesting service.
10. The use case terminates.

Figure 32: The use case describing how different actors create a vocabulary association in the INFOD registry.

A summary of steps describing the use case above are as follows:

1. The use case begins with the arrival of a request message.
2. The SOAP header is removed from the message.
3. The message is passed into an XML parser.
4. The manager validates the request message.
5. The manager inserts the new vocabulary association into the INFOD registry.
6. The manager creates a response message.
7. A SOAP header is added to the response message.
8. The manager returns the message to the requesting service.
9. The use case terminates.
A summary of steps describing the use case above are as follows:

1. The use case begins with the arrival of a request message.

2. The SOAP header is removed from the message.

3. The message is passed into an XML parser.

4. The manager validates the request message.

5. The manager searches for a list of elements that references this association.

6. The manager removes this association and all elements referencing this association from the INFOD registry.

7. The manager creates a response message.

8. A SOAP header is added to the response message.

9. The manager returns the message to the requesting service.
10. The use case terminates.

Figure 34: The use case describing how a subscriber creates a subscription in the INFOD registry.

A summary of steps describing the use case above are as follows:

1. The use case begins with the arrival of a request message.
2. The SOAP header is removed from the message.
3. The message is passed into an XML parser.
4. The manager validates the request message.
5. The manager inserts the new subscription into the INFOD registry.
6. The manager examines the new subscription and notifies the publishers who will generate messages for the subscription.
7. The notified publishers generate messages and send the messages to the consumers.
8. The manager creates a response message.
9. A SOAP header is added to the response message.

10. The manager returns the message to the requesting service.

11. The use case terminates.

Figure 35: The use case describing how a subscriber replaces a subscription in the INFOD registry.

A summary of steps describing the use case above are as follows:

1. The use case begins with the arrival of a request message.

2. The SOAP header is removed from the message.

3. The message is passed into an XML parser.

4. The manager validates the request message.

5. The manager replaces the subscription in the INFOD registry with the given EPR.

6. The manager examines the updated subscription and notifies the publishers who will generate messages for the subscription.
7. The notified publishers generate messages and send the messages to the consumers.

8. The manager creates a response message.

9. A SOAP header is added to the response message.

10. The manager returns the message to the requesting service.

11. The use case terminates.

Figure 36: The use case describing how a subscriber removes a subscription from the INFOD registry.

A summary of steps describing the use case above are as follows:

1. The use case begins with the arrival of a request message.

2. The SOAP header is removed from the message.

3. The message is passed into an XML parser.

4. The manager validates the request message.

5. The manager removes the subscription from the INFOD registry.

6. The manager notifies the publishers that a subscription is removed from the INFOD registry.
7. The manager creates a response message.

8. A SOAP header is added to the response message.

9. The manager returns the message to the requesting service.

10. The use case terminates.
A.2 Analyzing Use Cases to Identify Aspects

The following summarizes the analysis of the use cases used to identify the aspects of the INFOD system. Each use case categorizes its message flows in a base flow, an alternate flow and a subflow.

Use Case: Create Entities (Publishers, Consumers, Subscribers)

Basic Flow:
1. The use case begins with the arrival of a request message.
2. The SOAP header is removed from the message.
3. The message is passed into an XML Parser.
4. The manager validates the request.
5. The manager Insert Registry for the new entity.
6. The manager creates a response message.
7. A SOAP header is added to the response message.
8. The manager returns the message to the requesting service.
9. The use case terminates.

Sub Flows
S1. {subflow} XML Parser

Alternate Flows
A1. No Authorization on Creation
No Authorization occurs at step 5 of the base flow.
1. The manager checks if the user is authorized to add a new entity.
2. If not, the manager creates a fault message. (CreateEntityAuthorizationFailure).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

A2. Unknown Element Reference
Unknown Element Reference occurs at step 4 of the base flow.
1. The manager checks if all referenced elements in the request message already exists.
2. If not, the manager creates a fault message. (UnknownElementReferenceFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

A3. Missing Parameter
Missing Parameter occurs at step 4 of the base flow.
1. The manager checks if the request message has all required parameters.
2. If not, the manager creates a fault message. (MissingRequiredParameterFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

A4. Unsupported XQuery Expression
Unsupported XQuery Expression occurs at step 4 of the base flow.

1. The manager checks if the XQuery expression in the request message can be parsed correctly.
2. If not, the manager creates a fault message. (UnsupportedXQueryFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

Use Case: Replace Entities (Publishers, Consumers, Subscribers)

Basic Flow:
1. The use case begins with the arrival of a request message.
2. The SOAP header is removed from the message.
3. The message is passed into an XML Parser.
4. The manager validates the request.
5. The manager Update Registry using the given EPR.
6. The manager creates a response message.
7. A SOAP header is added to the response message.
8. The manager returns the message to the requesting service.
9. The use case terminates.

Sub Flows
S1. {subflow} XML Parser

Alternate Flows
A1. No Authorization on Replacement
No Authorization occurs at step 5 of the base flow.

1. The manager checks if the user is authorized to replace an entity.
2. If not, the manager creates a fault message. (ReplaceEntityAuthorizationFailure).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

A2. Unknown Element Reference
Unknown Element Reference occurs at step 4 of the base flow.

1. The manager checks if all referenced elements in the request message already exists.
2. If not, the manager creates a fault message. (UnknownElementReferenceFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

A3. Missing Parameter
Missing Parameter occurs at step 4 of the base flow.

1. The manager checks if the request message has all required parameters.
2. If not, the manager creates a fault message. (MissingRequiredParameterFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

A4. Unsupported XQuery Expression
Unsupported XQuery Expression occurs at step 4 of the base flow.

1. The manager checks if the XQuery expression in the request message can be parsed correctly.
2. If not, the manager creates a fault message. (UnsupportedXQueryFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

**Use Case: Drop Entities (Publishers, Consumers, Subscribers)**

**Basic Flow:**
1. The use case begins with the arrival of a request message.
2. The SOAP header is removed from the message.
3. The message is passed into an XML Parser.
4. The manager validates the request.
5. The manager creates a list of elements that references this entity.
6. The manager Drop Registry to remove this entity and all elements referencing this entity.
7. The manager creates a response message.
8. A SOAP header is added to the response message.
9. The manager returns the message to the requesting service.
10. The use case terminates.

**Sub Flows**

S1. (subflow) XML Parser

**Alternate Flows**

A1. No Authorization on Deletion
No Authorization occurs at step 6 of the base flow.
1. The manager checks if the user is authorized to drop an entity.
2. If not, the manager creates a fault message. (DropEntityAuthorizationFailure).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

A2. Unknown Element Reference
Unknown Element Reference occurs at step 4 of the base flow.
1. The manager checks if all referenced elements in the request message already exists.
2. If not, the manager creates a fault message. (UnknownElementReferenceFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

A3. Missing Parameter
Missing Parameter occurs at step 4 of the base flow.
1. The manager checks if the request message has all required parameters.
2. If not, the manager creates a fault message. (MissingRequiredParameterFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

A4. Invalid Execution Mode
Invalid Execution Mode occurs at step 4 of the base flow.
1. The manager checks if the specified execution mode is valid.
2. If not, the manager creates a fault message. (ExecutionModeFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.
A5. Drop Unused Entity
If in step 5 of the base flow, the specified option is “IF UNUSED”, the deletion will be done only if the entity is unused an unreferenced.
1. The manager checks the reference count of this entity.
2. The manager Drop Registry to remove the specified entity from the registry only if the count equals zero.
3. The subflow terminates.

A6. Disable New Entity
If in step 5 of the base flow, the specified option is “DISABLE NEW”, the deletion will be done only if the entity is unused an unreferenced.
1. The manager sets the disable flag of this entity.
2. The manager checks the reference count of this entity.
3. The manager Drop Registry to remove the specified entity from the registry only if the count equals zero.
4. The subflow terminates.

Use Case: Create Subscription

Basic Flow:
1. The use case begins with the arrival of a request message.
2. The SOAP header is removed from the message.
3. The message is passed into an XML Parser.
4. The manager validates the request.
5. The manager Insert Registry the new publisher.
7. The manager creates a response message.
8. A SOAP header is added to the response message.
9. The manager returns the message to the requesting service.
10. The use case terminates.

Sub Flows
S1. {subflow} XML Parser

Alternate Flows
A1. No Authorization on Creation
No Authorization occurs at step 5 of the base flow.
1. The manager checks if the user is authorized to add a new entity.
2. If not, the manager creates a fault message. (CreateEntityAuthorizationFailure).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

A2. Unknown Element Reference
Unknown Element Reference occurs at step 4 of the base flow.
1. The manager checks if all referenced elements in the request message already exists.
2. If not, the manager creates a fault message. (UnknownElementReferenceFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

A3. Missing Parameter
Missing Parameter occurs at step 4 of the base flow.
1. The manager checks if the request message has all required parameters.
2. If not, the manager creates a fault message. (MissingRequiredParameterFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

**A4. Unsupported XQuery Expression**
Unsupported XQuery Expression occurs at step 4 of the base flow.
1. The manager checks if the XQuery expression in the request message can be parsed correctly.
2. If not, the manager creates a fault message. (UnsupportedXQueryFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

**Use Case: Replace Subscription**

**Basic Flow:**
1. The use case begins with the arrival of a request message.
2. The SOAP header is removed from the message.
3. The message is passed into an XML Parser.
4. The manager validates the request.
5. The manager Update Registry using the given EPR.
7. The manager creates a response message.
8. A SOAP header is added to the response message.
9. The manager returns the message to the requesting service.
10. The use case terminates.

**Sub Flows**

**S1. {subflow} XML Parser**

**Alternate Flows**

**A1. No Authorization on Replacement**
No Authorization occurs at step 5 of the base flow.
1. The manager checks if the user is authorized to replace an entity.
2. If not, the manager creates a fault message. (ReplaceEntityAuthorizationFailure).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

**A2. Unknown Element Reference**
Unknown Element Reference occurs at step 4 of the base flow.
1. The manager checks if all referenced elements in the request message already exists.
2. If not, the manager creates a fault message. (UnknownElementReferenceFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

**A3. Missing Parameter**
Missing Parameter occurs at step 4 of the base flow.
1. The manager checks if the request message has all required parameters.
2. If not, the manager creates a fault message. (MissingRequiredParameterFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

A4. Unsupported XQuery Expression
Unsupported XQuery Expression occurs at step 4 of the base flow.
1. The manager checks if the XQuery expression in the request message can be parsed correctly.
2. If not, the manager creates a fault message. (UnsupportedXQueryFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

Use Case: Drop Subscription

Basic Flow:
1. The use case begins with the arrival of a request message.
2. The SOAP header is removed from the message.
3. The message is passed into an XML Parser.
4. The manager validates the request.
5. The manager creates a list of elements that references this entity.
6. The manager Drop Registry to remove this entity and all elements referencing this entity.
8. The manager creates a response message.
9. A SOAP header is added to the response message.
10. The manager returns the message to the requesting service.
11. The use case terminates.

Sub Flows
S1. [subflow] XML Parser

Alternate Flows
A1. No Authorization on Deletion
No Authorization occurs at step 6 of the base flow.
1. The manager checks if the user is authorized to drop an entity.
2. If not, the manager creates a fault message. (DropEntityAuthorizationFailure).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

A2. Unknown Element Reference
Unknown Element Reference occurs at step 4 of the base flow.
1. The manager checks if all referenced elements in the request message already exists.
2. If not, the manager creates a fault message. (UnknownElementReferenceFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

A3. Missing Parameter
Missing Parameter occurs at step 4 of the base flow.
1. The manager checks if the request message has all required parameters.
2. If not, the manager creates a fault message. (MissingRequiredParameterFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.
A4. Invalid Execution Mode
Invalid Execution Mode occurs at step 4 of the base flow.
1. The manager checks if the specified execution mode is valid.
2. If not, the manager creates a fault message. (ExecutionModeFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

A5. Drop Unused Entity
If in step 5 of the base flow, the specified option is “IF UNUSED”, the deletion will be done only if the entity is unused and unreferenced.
1. The manager checks the reference count of this entity.
2. The manager Drop Registry to remove the specified entity from the registry only if the count equals zero.
3. The subflow terminates.

A6. Disable New Entity
If in step 5 of the base flow, the specified option is “DISABLE NEW”, the deletion will be done only if the entity is unused and unreferenced.
1. The manager sets the disable flag of this entity.
2. The manager checks the reference count of this entity.
3. The manager Drop Registry to remove the specified entity from the registry only if the count equals zero.
4. The subflow terminates.

Use Case: Manage Registry

Base Flow
1. The use case begins with the arrival of a request message for querying data from the registry.
2. The SOAP header is removed from the message.
3. The message is passed into an XML Parser.
4. The manager validates the request.
5. The manager Search Registry with the given XPath expression.
6. The manager creates a response message.
7. A SOAP header is added to the response message.
8. The manager returns the message to the requesting service.
9. The use case terminates.

Subflow:
S1. Insert Registry
1. The manager creates a new node to be inserted into the registry.
2. The manager fills the values according to the given data.
3. The manager creates a reference count for new node.
4. The manager updates the reference counts of other nodes referenced by this new node.
5. The manager inserts the new node to the registry.
6. The manager generates a unique EPR for this new node.
7. The manager returns the EPR.
8. The subflow terminates.

S2. Update Registry
1. The manager finds the node in the registry with the given EPR.
2. The manager updates the values according to the given data.
3. The manager updates the node in the registry.
4. The subflow terminates.

S3. Drop Registry
1. The manager finds the node in the registry with the given EPR.
2. The manager removes the node from the registry.
3. The manager commits the deletion.
4. The subflow terminates.

S4. Search Registry
1. The manager loads the XML document (e.g. Registry.xml).
2. The manager evaluates the XPath expression.
3. The manager returns the data.

Alternate Flows
A1. No Authorization to Query Data
No Authorization occurs at step 5 of the base flow.
1. The manager checks if the user is authorized to query the registry.
2. If not, the manager creates a fault message. (GetMetaDataAuthorizationFailure).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

A2. Missing Parameter
Missing Parameter occurs at step 4 of the base flow.
1. The manager checks if the request message has all required parameters.
2. If not, the manager creates a fault message. (MissingRequiredParameterFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

A3. Unsupported XQuery Expression
Unsupported XQuery Expression occurs at step 4 of the base flow.
1. The manager checks if the XQuery expression in the request message can be parsed correctly.
2. If not, the manager creates a fault message. (UnsupportedXQueryFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

Use Case: Message Publication

Subflow
S1. Notify Publishers
1. The manager Search Registry for the data that are interested to the given message.
2. The manager Search Registry for a list of association using the data vocabulary EPR found in the message.
3. The manager Search Registry for a list of publishers to be notified from the list of associations.
4. The manager Filter the publishers specified in the subscription.
5. The manager Search Registry for the list of consumers that are interested to the data.
6. The manager Filter the consumers specified in the associations.
7. The manager Create Message.
S2. Create Message
1. The manager Filter the consumers specified in the constraints of the eligible publishers.
2. The manager creates a message for each eligible publisher which references the interested data.
3. The manager Notify Consumers on behalf of the eligible publishers.
4. The subflow terminates.

S3. Notify Consumers
1. The manager receives a message sent by a publisher.
2. The manager removes the SOAP header from the message.
3. The message is passed to the XML parser.
4. The manager Filter the message.
5. The manager propagates the data to the actual service on behalf of the consumer entity.

S4. Filter
1. The manager checks if the elements referenced in the given message matches one of the constraints of the given entity.
2. The manager gets a list of elements from the registry that matches the constraints of the given entity.
3. The manager removes the elements that are not contained in the restriction list.
4. The manager returns the resulting list.
5. This subflow terminates.

Use Case: Registry Property Vocabularies

Basic Flow:
1. The use case begins with the arrival of a request message.
2. The SOAP header is removed from the message.
3. The message is passed into an XML Parser.
4. The manager validates the request.
5. The manager Insert Registry the new vocabulary.
6. The manager creates a response message.
7. A SOAP header is added to the response message.
8. The manager returns the message to the requesting service.
9. The use case terminates.

Sub Flows
S1. {subflow} XML Parser

Alternate Flows
A1. No Authorization on Creation
No Authorization occurs at step 5 of the base flow.
1. The manager checks if the user is authorized to add a new vocabulary.
2. If not, the manager creates a fault message. (RegisterVocabularyAuthorizationFailure).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

A2. Unknown Element Reference
Unknown Element Reference occurs at step 4 of the base flow.
1. The manager checks if all referenced elements in the request message already exists.
2. If not, the manager creates a fault message. (UnknownElementReferenceFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

A3. Missing Parameter
Missing Parameter occurs at step 4 of the base flow.
1. The manager checks if the request message has all required parameters.
2. If not, the manager creates a fault message. (MissingRequiredParameterFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

A4. Unsupported Vocabulary Fault
Unsupported Vocabulary Fault occurs at step 4 of the base flow.
1. The manager checks if the vocabulary language specified in the request message is supported.
2. If not, the manager creates a fault message. (UnSupportedVocabularyFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

Use Case: Registry Data Vocabularies

Basic Flow:
1. The use case begins with the arrival of a request message.
2. The SOAP header is removed from the message.
3. The message is passed into an XML Parser.
4. The manager validates the request.
5. The manager Insert Registry the new vocabulary.
6. The manager creates a response message.
7. A SOAP header is added to the response message.
8. The manager returns the message to the requesting service.
9. The use case terminates.

Sub Flows
S1. {subflow} XML Parser

Alternate Flows
A1. No Authorization on Creation
No Authorization occurs at step 5 of the base flow.
1. The manager checks if the user is authorized to add a new vocabulary.
2. If not, the manager creates a fault message. (RegisterVocabularyAuthorizationFailure).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

A2. Unknown Element Reference
Unknown Element Reference occurs at step 4 of the base flow.
1. The manager checks if all referenced elements in the request message already exists.
2. If not, the manager creates a fault message. (UnknownElementReferenceFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

A3. Missing Parameter
Missing Parameter occurs at step 4 of the base flow.
1. The manager checks if the request message has all required parameters.
2. If not, the manager creates a fault message. (MissingRequiredParameterFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

A4. Unsupported Vocabulary Fault
Unsupported Vocabulary Fault occurs at step 4 of the base flow.
1. The manager checks if the vocabulary language specified in the request message is supported.
2. If not, the manager creates a fault message. (UnSupportedVocabularyFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

Use Case: Unregister Vocabulary

Basic Flow:
1. The use case begins with the arrival of a request message.
2. The SOAP header is removed from the message.
3. The message is passed into an XML Parser.
4. The manager validates the request.
5. The manager creates a list of elements that references this vocabulary.
6. The manager Drop Registry to remove this entity and all elements referencing this entity.
7. The manager creates a response message.
8. A SOAP header is added to the response message.
9. The manager returns the message to the requesting service.
10. The use case terminates.

Sub Flows
S1. {subflow} XML Parser

Alternate Flows
A1. No Authorization on Deletion
No Authorization occurs at step 6 of the base flow.
1. The manager checks if the user is authorized to unregister a vocabulary.
2. If not, the manager creates a fault message. (UnregisterVocabularyAuthorizationFailure).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

A2. Unknown Element Reference
Unknown Element Reference occurs at step 4 of the base flow.
1. The manager checks if all referenced elements in the request message already exists.
2. If not, the manager creates a fault message. (UnknownElementReferenceFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

A3. Missing Parameter
Missing Parameter occurs at step 4 of the base flow.
1. The manager checks if the request message has all required parameters.
2. If not, the manager creates a fault message. (MissingRequiredParameterFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

A4. Invalid Execution Mode
Invalid Execution Mode occurs at step 4 of the base flow.
1. The manager checks if the specified execution mode is valid.
2. If not, the manager creates a fault message. (ExecutionModeFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

A5. Drop Unused Vocabulary
If in step 6 of the base flow, the specified option is “IF UNUSED”, the deletion will be done only if the vocabulary is unused and unreferenced.
1. The manager checks the reference count of this vocabulary.
2. The manager Drop Registry to remove the specified vocabulary from the registry only if the count equals zero.
3. The subflow terminates.

A6. Disable New Vocabulary
If in step 6 of the base flow, the specified option is “DISABLE NEW”, the deletion will be done only if the vocabulary is unused and unreferenced.
1. The manager sets the disable flag of this entity.
2. The manager checks the reference count of this vocabulary.
3. The manager Drop Registry to remove the specified vocabulary from the registry only if the count equals zero.
4. The subflow terminates.

Use Case: Create Property Vocabulary Instances

Basic Flow:
1. The use case begins with the arrival of a request message.
2. The SOAP header is removed from the message.
3. The message is passed into an XML Parser.
4. The manager validates the request.
5. The manager Insert Registry the new vocabulary instance.
6. The manager creates a response message.
7. A SOAP header is added to the response message.
8. The manager returns the message to the requesting service.
9. The use case terminates.

Sub Flows
S1. {subflow} XML Parser

Alternate Flows
A1. No Authorization on Creation
No Authorization occurs at step 5 of the base flow.
1. The manager checks if the user is authorized to add a new vocabulary instance.
2. If not, the manager creates a fault message. (CreatePropertyVocabularyInstanceAuthorizationFailure).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

A2. Unknown Element Reference
Unknown Element Reference occurs at step 4 of the base flow.
1. The manager checks if all referenced elements in the request message already exists.
2. If not, the manager creates a fault message. (UnknownElementReferenceFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

A3. Missing Parameter
Missing Parameter occurs at step 4 of the base flow.
1. The manager checks if the request message has all required parameters.
2. If not, the manager creates a fault message. (MissingRequiredParameterFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

A4. Unsupported Vocabulary Fault
Unsupported Vocabulary Fault occurs at step 4 of the base flow.
1. The manager checks if the vocabulary language specified in the request message is supported.
2. If not, the manager creates a fault message. (UnSupportedVocabularyFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

Use Case: Drop Property Vocabulary Instances

Basic Flow:
1. The use case begins with the arrival of a request message.
2. The SOAP header is removed from the message.
3. The message is passed into an XML Parser.
4. The manager validates the request.
5. The manager creates a list of elements that references this vocabulary instance.
6. The manager Drop Registry to remove this instance and all elements referencing this instance.
7. The manager creates a response message.
8. A SOAP header is added to the response message.
9. The manager returns the message to the requesting service.
10. The use case terminates.

Sub Flows
S1. {subflow} XML Parser

Alternate Flows
A1. No Authorization on Deletion
No Authorization occurs at step 6 of the base flow.
1. The manager checks if the user is authorized to remove the vocabulary instance.
2. If not, the manager creates a fault message. (DropPropertyVocabularyInstanceAuthorizationFailure).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.
A2. Unknown Element Reference
Unknown Element Reference occurs at step 4 of the base flow.
1. The manager checks if all referenced elements in the request message already exists.
2. If not, the manager creates a fault message. (UnknownElementReferenceFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

A3. Missing Parameter
Missing Parameter occurs at step 4 of the base flow.
1. The manager checks if the request message has all required parameters.
2. If not, the manager creates a fault message. (MissingRequiredParameterFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

A4. Invalid Execution Mode
Invalid Execution Mode occurs at step 4 of the base flow.
1. The manager checks if the specified execution mode is valid.
2. If not, the manager creates a fault message. (ExecutionModeFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

A5. Drop Unused Vocabulary Instance
If in step 6 of the base flow, the specified option is “IF UNUSED”, the deletion will be done only if the instance is unused and unreferenced.
1. The manager checks the reference count of this vocabulary instance.
2. The manager Drop Registry to remove the specified vocabulary instance from the registry only if the count equals zero.
3. The subflow terminates.

A6. Disable New Vocabulary Instance
If in step 6 of the base flow, the specified option is “DISABLE NEW”, the deletion will be done only if the instance is unused and unreferenced.
1. The manager sets the disable flag of this instance.
2. The manager checks the reference count of this vocabulary instance.
3. The manager Drop Registry to remove the specified vocabulary instance from the registry only if the count equals zero.
4. The subflow terminates.

Use Case: Associate Vocabularies

Basic Flow:
1. The use case begins with the arrival of a request message.
2. The SOAP header is removed from the message.
3. The message is passed into an XML Parser.
4. The manager validates the request.
5. The manager Insert Registry the new association.
6. The manager creates a response message.
7. A SOAP header is added to the response message.
8. The manager returns the message to the requesting service.
9. The use case terminates.
**Sub Flows**

S1. *(subflow) XML Parser*

### Alternate Flows

**A1. No Authorization on Associate Vocabulary**

No Authorization occurs at step 5 of the base flow.

1. The manager checks if the user is authorized to add a new association.
2. If not, the manager creates a fault message. *(CreateAssociationAuthorizationFailure)*.
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

**A2. Unknown Element Reference**

Unknown Element Reference occurs at step 4 of the base flow.

1. The manager checks if all referenced elements in the request message already exists.
2. If not, the manager creates a fault message. *(UnknownElementReferenceFault)*.
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

**A3. Missing Parameter**

Missing Parameter occurs at step 4 of the base flow.

1. The manager checks if the request message has all required parameters.
2. If not, the manager creates a fault message. *(MissingRequiredParameterFault)*.
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

**A4. Unsupported XQuery Expression**

Unsupported XQuery Expression occurs at step 4 of the base flow.

1. The manager checks if the XQuery expression in the request message can be parsed correctly.
2. If not, the manager creates a fault message. *(UnsupportedXQueryFault)*.
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

**Use Case: Disassociate Vocabularies**

**Basic Flow:**

1. The use case begins with the arrival of a request message.
2. The SOAP header is removed from the message.
3. The message is passed into an XML Parser.
4. The manager validates the request.
5. The manager creates a list of elements that references this association.
6. The manager Drop Registry to remove this association and all elements referencing this association.
7. The manager creates a response message.
8. A SOAP header is added to the response message.
9. The manager returns the message to the requesting service.
10. The use case terminates.

**Sub Flows**
**S1. {subflow} XML Parser**

**Alternate Flows**

**A1. No Authorization on Disassociate Vocabulary**
No Authorization occurs at step 6 of the base flow.
1. The manager checks if the user is authorized to remove the association.
2. If not, the manager creates a fault message. (DisAssociationAuthorizationFailure).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

**A2. Unknown Element Reference**
Unknown Element Reference occurs at step 4 of the base flow.
1. The manager checks if all referenced elements in the request message already exists.
2. If not, the manager creates a fault message. (UnknownElementReferenceFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

**A3. Missing Parameter**
Missing Parameter occurs at step 4 of the base flow.
1. The manager checks if the request message has all required parameters.
2. If not, the manager creates a fault message. (MissingRequiredParameterFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

**A4. Invalid Execution Mode**
Invalid Execution Mode occurs at step 4 of the base flow.
1. The manager checks if the specified execution mode is valid.
2. If not, the manager creates a fault message. (ExecutionModeFault).
3. A SOAP header is added to the fault message.
4. The manager returns the message to the requesting service.
5. The base flow terminates.

**A5. Drop Unused Association**
If in step 6 of the base flow, the specified option is “IF UNUSED”, the deletion will be done only if the association is unused and unreferenced.
1. The manager checks the reference count of this association.
2. The manager **Drop Registry** to remove the specified association from the registry only if the count equals zero.
3. The subflow terminates.

**A6. Disable New Association**
If in step 6 of the base flow, the specified option is “DISABLE NEW”, the deletion will be done only if the association is unused and unreferenced.
1. The manager sets the disable flag of this association.
2. The manager checks the reference count of this association.
3. The manager **Drop Registry** to remove the specified association from the registry only if the count equals zero.
4. The subflow terminates.
A.3 Makefile Example

Below is one of the Makefiles generated by the Perl script.

CC = tacc
AR = accar
RANLIB = ranlib
CONFIG=xml2-config
CFLAGS=--cflags
LIBS=--libs
OPTION=--w

###
# testcase files
###
TESTCASE=testcase_publisher.c testcase_consumer.c testcase_subscriber.c testcase_propvocab.c
testcase_datavocab.c testcase_association.c testcase_subscription.c testcase_getMetaData.c
###
# core files
###
CORE=publisherManager.c consumerManager.c subscriberManager.c subscriptionManager.c
associationManager.c registryManager.c vocabularyManager.c utilities.c xmlparse.c
COREOBJ=publisherManager.o consumerManager.o subscriberManager.o subscriptionManager.o
associationManager.o registryManager.o vocabularyManager.o utilities.o xmlparse.o
CORELIB=-L . -linfod
CORELIBFILE=libinfod.a
###
# aspect files
###
ASPECT=detachSoap.acc exchandler.acc filter.acc logging.acc removeReferences.acc disableNew.acc
removeUnused.acc namespace.acc attachSoap.acc
TCASPECT=unittest_detachSoap.acc unittest_exc.acc unittest_filter.acc unittest_cascade.acc
unittest_disableNew.acc unittest_ifunused.acc unittest_namespace.acc unittest_attachSoap.acc

all: copy build test clean

$(CORELIBFILE): $(CORE) $(ASPECT) $(TCASPECT)
    $(CC) -c $(OPTION) $(CONFIG) $(CFLAGS) $(CORE) $(ASPECT) $(TCASPECT)
    $(AR) rc $(CORELIBFILE) $(COREOBJ)
    $(RANLIB) $(CORELIBFILE)

build: $(TESTCASE) $(CORELIBFILE) $(TCASPECT)
    $(CC) -c $(OPTION) $(CONFIG) $(CFLAGS) testcase_publisher.c $(TCASPECT)
    $(CC) -o testcase_publisher $(OPTION) $(CONFIG) $(CFLAGS) testcase_publisher.o
    $(CORELIB) $(CONFIG) $(LIBS)
        $(CC) -c $(OPTION) $(CONFIG) $(CFLAGS) testcase_consumer.c $(TCASPECT)
        $(CC) -o testcase_consumer $(OPTION) $(CONFIG) $(CFLAGS) testcase_consumer.o
    $(CORELIB) $(CONFIG) $(LIBS)
        $(CC) -c $(OPTION) $(CONFIG) $(CFLAGS) testcase_subscriber.c $(TCASPECT)
        $(CC) -o testcase_subscriber $(OPTION) $(CONFIG) $(CFLAGS) testcase_subscriber.o
    $(CORELIB) $(CONFIG) $(LIBS)
        $(CC) -c $(OPTION) $(CONFIG) $(CFLAGS) testcase_propvocab.c $(TCASPECT)
$(CC) -o testcase_propvocab $(OPTION) `$(CONFIG) $(CFLAGS)` testcase_propvocab.o
$(CORELIB) `$(CONFIG) $(LIBS)`
$(CC) -c $(OPTION) `$(CONFIG) $(CFLAGS)` testcase_datavocab.c $(TCASPECT)
$(CC) -o testcase_datavocab $(OPTION) `$(CONFIG) $(CFLAGS)` testcase_datavocab.o
$(CORELIB) `$(CONFIG) $(LIBS)`
$(CC) -c $(OPTION) `$(CONFIG) $(CFLAGS)` testcase_vocabinst.c $(TCASPECT)
$(CC) -o testcase_vocabinst $(OPTION) `$(CONFIG) $(CFLAGS)` testcase_vocabinst.o
$(CORELIB) `$(CONFIG) $(LIBS)`
$(CC) -c $(OPTION) `$(CONFIG) $(CFLAGS)` testcase_association.c $(TCASPECT)
$(CC) -o testcase_association $(OPTION) `$(CONFIG) $(CFLAGS)` testcase_association.o
$(CORELIB) `$(CONFIG) $(LIBS)`
$(CC) -c $(OPTION) `$(CONFIG) $(CFLAGS)` testcase_subscription.c $(TCASPECT)
$(CC) -o testcase_subscription $(OPTION) `$(CONFIG) $(CFLAGS)` testcase_subscription.o
$(CORELIB) `$(CONFIG) $(LIBS)`
$(CC) -c $(OPTION) `$(CONFIG) $(CFLAGS)` testcase_getMetaData.c $(TCASPECT)
$(CC) -o testcase_getMetaData $(OPTION) `$(CONFIG) $(CFLAGS)` testcase_getMetaData.o
$(CORELIB) `$(CONFIG) $(LIBS)`

test:
  ./testcase_publisher add
  ./testcase_consumer add
  ./testcase Subscriber add
  ./testcase_propvocab register
  ./testcase_datavocab register
  ./testcase_vocabinst add
  ./testcase_association add
  ./testcase_subscription add
  ./testcase_getMetaData
  ./testcase_publisher drop
  ./testcase_consumer drop
  ./testcase Subscriber drop
  ./testcase_propvocab unregister
  ./testcase_datavocab unregister
  ./testcase_vocabinst drop
  ./testcase_association drop
  ./testcase_subscription drop

copy:
  cp -f ./*.c .
  cp -f ./*.h .
  cp -f ./*.acc .
  cp -f ./infodRegistry.xml .
  cp -f ./users.xml .

server: $(CORE) $(ASPECT) performance.acc server.c
  $(CC) -o server $(OPTION) `$(CONFIG) $(CFLAGS)` server.c $(CORE) $(ASPECT)
  performance.acc `$(CONFIG) $(LIBS)`

client: $(TESTCASE) $(CORELIBFILE) client.acc
  $(CC) -c $(OPTION) `$(CONFIG) $(CFLAGS)` testcase_publisher.c client.acc
  $(CC) -o testcase_publisher $(OPTION) `$(CONFIG) $(CFLAGS)` testcase_publisher.o
  $(CORELIB) `$(CONFIG) $(LIBS)`
  $(CC) -c $(OPTION) `$(CONFIG) $(CFLAGS)` testcase_consumer.c client.acc
  $(CC) -o testcase_consumer $(OPTION) `$(CONFIG) $(CFLAGS)` testcase_consumer.o
  $(CORELIB) `$(CONFIG) $(LIBS)`
  $(CC) -c $(OPTION) `$(CONFIG) $(CFLAGS)` testcase_subscriber.c client.acc
$(CC) -o testcase_subscriber $(OPTION) `$(CONFIG) $(CFLAGS)` testcase_subscriber.o
$(CORELIB) `$(CONFIG) $(LIBS)`
$(CC) -c $(OPTION) `$(CONFIG) $(CFLAGS)` testcase_propvocab.c client.acc
$(CC) -o testcase_propvocab $(OPTION) `$(CONFIG) $(CFLAGS)` testcase_propvocab.o
$(CORELIB) `$(CONFIG) $(LIBS)`
$(CC) -c $(OPTION) `$(CONFIG) $(CFLAGS)` testcase_datavocab.c client.acc
$(CC) -o testcase_datavocab $(OPTION) `$(CONFIG) $(CFLAGS)` testcase_datavocab.o
$(CORELIB) `$(CONFIG) $(LIBS)`
$(CC) -c $(OPTION) `$(CONFIG) $(CFLAGS)` testcase_vocabinst.c client.acc
$(CC) -o testcase_vocabinst $(OPTION) `$(CONFIG) $(CFLAGS)` testcase_vocabinst.o
$(CORELIB) `$(CONFIG) $(LIBS)`
$(CC) -c $(OPTION) `$(CONFIG) $(CFLAGS)` testcase_association.c client.acc
$(CC) -o testcase_association $(OPTION) `$(CONFIG) $(CFLAGS)` testcase_association.o
$(CORELIB) `$(CONFIG) $(LIBS)`
$(CC) -c $(OPTION) `$(CONFIG) $(CFLAGS)` testcase_subscription.c client.acc
$(CC) -o testcase_subscription $(OPTION) `$(CONFIG) $(CFLAGS)` testcase_subscription.o
$(CORELIB) `$(CONFIG) $(LIBS)`
$(CC) -c $(OPTION) `$(CONFIG) $(CFLAGS)` testcase_getMetaData.c client.acc
$(CC) -o testcase_getMetaData $(OPTION) `$(CONFIG) $(CFLAGS)` testcase_getMetaData.o
$(CORELIB) `$(CONFIG) $(LIBS)`

remote: server client

clean:
    rm -f *_t1.*
    rm -f *_t2.*
    rm -f *.bak
    rm -f *.stackdump
    rm -f *.o
    rm -f lib*.a
    rm -fR .acc_dir
    rm -f testcase_*.exe
### A.4 Test Data Example

The following is some test data copied from the large data set to do the performance evaluation:

```xml
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
          xmlns:infod="http://www.ggf.com/infod">
  <soap:Header>
    <username>flannanl</username>
  </soap:Header>
  <infod:CreateSubscription
          xmlns:infod="http://www.ggf.com/infod"
    <infod:SubscriptionName>CarDealerRequest0</infod:SubscriptionName>
    <infod:SubscriptionDescription>My car choice</infod:SubscriptionDescription>
    <infod:SubscriberReference>
      <wsa:Address>http://www.example.org/someSubscriberEPR</wsa:Address>
    </infod:SubscriberReference>
    <infod:DataConstraints>//datavocabularies/CarInventorYEPR</infod:DataConstraints>
  </infod:CreateSubscription>
</soap:Envelope>

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
          xmlns:infod="http://www.ggf.com/infod">
  <soap:Header>
    <username>flannanl</username>
  </soap:Header>
  <infod:CreateSubscription
          xmlns:infod="http://www.ggf.com/infod"
    <infod:SubscriptionName>CarDealerRequest1</infod:SubscriptionName>
    <infod:SubscriptionDescription>My car choice</infod:SubscriptionDescription>
    <infod:SubscriberReference>
      <wsa:Address>http://www.example.org/someSubscriberEPR</wsa:Address>
    </infod:SubscriberReference>
    <infod:DataConstraints>//datavocabularies/CarInventorYEPR</infod:DataConstraints>
  </infod:CreateSubscription>
</soap:Envelope>

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
          xmlns:infod="http://www.ggf.com/infod">
  <soap:Header>
    <username>flannanl</username>
  </soap:Header>
  <infod:CreateSubscription
          xmlns:infod="http://www.ggf.com/infod"
    <infod:SubscriptionName>CarDealerRequest2</infod:SubscriptionName>
    <infod:SubscriptionDescription>My car choice</infod:SubscriptionDescription>
    <infod:SubscriberReference>
      <wsa:Address>http://www.example.org/someSubscriberEPR</wsa:Address>
    </infod:SubscriberReference>
    <infod:DataConstraints>//datavocabularies/CarInventorYEPR</infod:DataConstraints>
  </infod:CreateSubscription>
</soap:Envelope>
```
A.5 README to Configure INFOD System and Run Test Cases

This README describes how to configure the INFOD system and also how to run the test cases on the INFOD system.

To build one configuration, do the following:

1. Extract the INFOD code from the CVS Repository.
2. Export ACCHOME to your ACC install directory.
3. Change the working directory to the <Install Dir>/infod directory.
4. Run 'perl make.pl' on the command line.
   This will create a Makefile for each configuration.
5. Run './build utestXXXX' where utestXXXX is the directory containing the Makefile.
   This will build the libinfod.a in the utestXXXX directory.

To build all configurations, do the following:

1. Extract the INFOD code from the CVS Repository.
2. Export ACCHOME to your ACC install directory.
3. Change the working directory to the <Install Dir>/infod directory.
4. Run 'perl make.pl' on the command line.
   This will create a Makefile for each configuration.
5. Run './build utest*' on the command line.
   This will build the libinfod.a in all utestXXXX directories.

To run all test cases, do the following:

1. Extract the INFOD code from the CVS Repository.
2. Export ACCHOME to your ACC install directory.
3. Change the working directory to the <Install Dir>/infod directory.
4. Run 'make all' on the command line.
   'make all' will execute the perl script to create a Makefile for each configuration and then kick off the tests.

To run one test case, do the following:

1. Extract the INFOD code from the CVS Repository.
2. Export ACCHOME to your ACC install directory.
3. Change the working directory to the <Install Dir>/infod directory.
4. Run 'perl make.pl' on the command line.
   This will create a Makefile for each configuration.
5. Run './runTest utestXXXX' where utestXXXX is the directory containing the Makefile.
   This will run the test case specified on the command line.

Special Notes:

On Cygwin, the InterProcess Communication (IPC) APIs are disabled by default. These APIs are required by the performance aspect to enable semaphores and access to the INFOD registry in the shared memory. To enable these APIs, run '/usr/sbin/cygserver' on the command line.
A.6 Libxml2

The following standards are implemented by Libxml2:

- the XML standard: [http://www.w3.org/TR/REC-xml](http://www.w3.org/TR/REC-xml)
- Namespaces in XML: [http://www.w3.org/TR/REC-xml-names/](http://www.w3.org/TR/REC-xml-names/)
- XML Base: [http://www.w3.org/TR/xmlbase/](http://www.w3.org/TR/xmlbase/)
- **RFC 2396**: Uniform Resource Identifiers
- XML Path Language (XPath) 1.0: [http://www.w3.org/TR/xpath](http://www.w3.org/TR/xpath)
- HTML4 parser: [http://www.w3.org/TR/html401/](http://www.w3.org/TR/html401/)
- XML Pointer Language (XPointer) Version 1.0: [http://www.w3.org/TR/xptr](http://www.w3.org/TR/xptr)
- XML Inclusions (XInclude) Version 1.0: [http://www.w3.org/TR/xinclude/](http://www.w3.org/TR/xinclude/)
- ISO-8859-x encodings, as well as [rfc2044](http://www.ietf.org/rfc/rfc2044.txt) [UTF-8] and [rfc2781](http://www.ietf.org/rfc/rfc2781.txt) [UTF-16]
  Unicode encodings, and more if using iconv support
- part of SGML Open Technical Resolution TR9401:1997
- XML Catalogs Working Draft 06 August 2001: [http://www.oasis-open.org/committees/entity/spec-2001-08-06.html](http://www.oasis-open.org/committees/entity/spec-2001-08-06.html)
- Canonical XML Version 1.0: [http://www.w3.org/TR/xml-c14n](http://www.w3.org/TR/xml-c14n) and the Exclusive XML Canonicalization CR draft [http://www.w3.org/TR/xml-exc-c14n](http://www.w3.org/TR/xml-exc-c14n)
- W3C XML Schemas Part 2: Datatypes [REC 02 May 2001](http://www.w3.org/TR/REC-xml)


• W3C \textit{xml:id} Working Draft 7 April 2004