

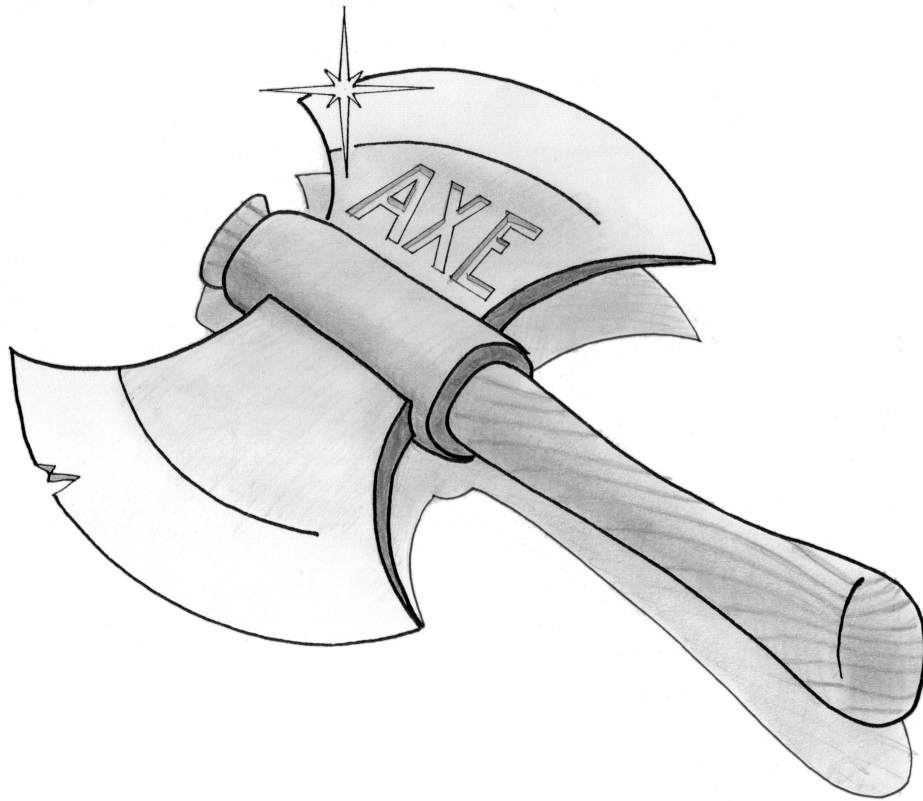
---

# Creating a Plug-in Collecting to AXE

(SW5-REPORT BY GROUP S504A)

*Application Development*

---



Dec 21<sup>th</sup>, 2004

AALBORG UNIVERSITY



# Department of Computer Science

Aalborg University

**Title:**

The Implementation of AXE  
- *The Plug-in Collection to AXE*

**Project period:**

SW5, Sep 2<sup>nd</sup> - Dec 21<sup>th</sup>, 2004

**Semester topic:**

Application Development

**Project group:**

E2-208

**Participants:**

---

Kristian Bødtker

---

Dan Lund Christensen

---

Martin Madsen

---

Peter Sørensen

---

Mads Vøge

**Supervisor:**

Linas Bukauskas

**Copies made:** 8**Number of pages:** 114**Project submitted:** Dec 21<sup>th</sup>, 2004**Appendix (pages):**

26

**ABSTRACT:**

AXE is short for Aalborg University XML Editor, which is an extendable XML editor capable of handling large files.

The development of AXE is a collaborative work done by four development groups, where each group has developed a specific part of the overall system.

Developing an XML editor was motivated by the fact that – at the time of development – no good, free XML editor existed. Relating this to the fact that XML plays an important role in many areas of communication and data management, it seemed as an area worthy of contribution.

AXE is an extendable editor platform, featuring an XML editor with syntax highlighting, clear text searching, version control system and well-formedness checking and validation of XML documents.

This report focuses on the work done by one of the four groups. It describes the analysis, design and implementation of a plug-in collection featuring well-formedness checking, validation, XPath querying and DTD to XML Schema conversion.



# Preface

## The Official Requirements

The project was met with requirements from the semester coordinator and with requirements outlined in the study regulation for this semester. The requirements fall into two categories: Requirements for the project, and requirements for the report.

The requirements for the project are subdivided into requirements for the overall project and requirements for the individual projects. The requirements for the overall project are marked out in the study regulation. The study regulation says, that:

The student must:

- Achieve insight into techniques that are central to the development of applications, that solve realistic problems.
- Achieve skills in analyzing, designing, programming and testing an application in a technical or organizational environment.

The requirements for the individual project are given in the guidelines from the semester coordinator. Among these, the main requirement is that the project must make use of the semesters two PE courses, which are System Analysis and Design (SAD) and Advanced Algorithm Design and Analysis (AALG).

The requirements for the report is also subdivided into two sets of requirements. One concerning the documentation of the analysis and design of the overall project, and one concerning the documentation of the individual projects. The requirements specify that each individual report must document the analysis and design of their individually assigned part of the project.

## How the Report is Organized

**Chapter 1 - Introduction** Contains a short description of what we hope to achieve in this report. It also gives a short introduction to how the multiproject is organized.

**Chapter 2 - Joint Analysis and Design** Introduces what the overall goal of the multiproject is. The material is provided by the joint work between all the participating groups.

**Chapter 3 - Analysis** Documents the analysis phase of the project by giving class-diagrams and state-charts.

**Chapter 4 - Design** Documents the design of the individual classes of the system.

**Chapter 5 - Implementation** Documents the implementation of the project.

**Chapter 6 - Continuous Checking of Large Files** Documents an approach to do continuous checking of large XML documents.

**Chapter 8 - Further development** Contains a discussion of the further potential of the project.

**Chapter 7 - Testing** Describes the testing phase of the project.

**Chapter 9 - Process** Contains a discussion of what went well or wrong in managing a multiproject, and a recommendation for what could be done to rectify the encountered problems.

**Chapter 10 - Conclusion** About what was learned and new insight acquired during the work with this project.

**Appendix A - System Requirements Specification** Contains the system requirement specification for the overall system.

**Appendix B - Protocols** Contains the protocols used in the collaborative work as an agreement on how to do things.

**Appendix C - DTD** Contains an example of a Document Type Definition (DTD)

**Appendix D - XML-Schema** Contains an example of an XML Schema

**Appendix E - testDTD.xml** Contains a DTD used for testing.

**Appendix F - testXSD.xml** Contains a Schema used for testing.

## Conventions Used in This Report

### Standard Notation

Throughout the report standard UML notation is used. If non-standard UML notation is used, it is clearly stated where the notation used differs from the standard.

### Emphasizing Conventions

The following font conventions are used in the report:

*Italic* is used for:

- events
- functions

- properties

Sans Serif is used for:

- Classes
- Clusters
- States

*Italic Sans Serif* is used for:

- Abstract Classes

Constant width is used for:

- source code

**Constant width bold face** is used for emphasis within source code.

When larger section of source code is shown, the following listings style is used.

```
1 class Hello
  {
    static void Main()
    {
5     System.Console.WriteLine("Hello World");
    }
  }
```

Listing 1: source example



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Collaborative Work . . . . .	1
1.2	XML . . . . .	2
1.3	The Task . . . . .	2
1.4	Development Method . . . . .	2
<b>2</b>	<b>Joint Analysis and Design</b>	<b>3</b>
2.1	Interview . . . . .	3
2.2	Delegation of Responsibility . . . . .	3
<b>3</b>	<b>Analysis</b>	<b>5</b>
3.1	The Task . . . . .	5
3.1.1	Purpose . . . . .	5
3.1.2	System Definition . . . . .	7
3.1.3	Context . . . . .	8
3.2	Cluster Structure of the the Plug-in Collection . . . . .	10
3.3	Common . . . . .	11
3.3.1	Structure . . . . .	11
3.3.2	Behavior . . . . .	12
3.3.3	Usage . . . . .	13
3.3.4	Functions . . . . .	15
3.4	Well-formedness . . . . .	17
3.4.1	Structure . . . . .	17
3.4.2	Behavior . . . . .	17
3.4.3	Usage . . . . .	18
3.4.4	Functions . . . . .	18

3.4.5	User Interface . . . . .	19
3.5	Validation . . . . .	20
3.5.1	Structure . . . . .	20
3.5.2	Behavior . . . . .	20
3.5.3	Usage . . . . .	21
3.5.4	Functions . . . . .	22
3.5.5	User Interface . . . . .	22
3.6	XPath . . . . .	23
3.6.1	Structure . . . . .	23
3.6.2	Behavior . . . . .	23
3.6.3	Usage . . . . .	24
3.6.4	Functions . . . . .	24
3.6.5	User Interface . . . . .	25
3.7	DTD2Schema . . . . .	26
3.7.1	Behavior . . . . .	26
3.7.2	Usage . . . . .	27
3.7.3	Functions . . . . .	27
3.7.4	User Interface . . . . .	27
<b>4</b>	<b>Design</b>	<b>29</b>
4.1	Corrections and Assumptions to the Analysis . . . . .	29
4.1.1	Corrections . . . . .	29
4.1.2	Assumptions . . . . .	29
4.2	Quality Attributes . . . . .	30
4.3	The Technical Surroundings . . . . .	33
4.3.1	Equipment . . . . .	33
4.3.2	Programs Needed . . . . .	33
4.4	Architecture . . . . .	34
4.4.1	Component Architecture . . . . .	34
4.5	Common Component . . . . .	36
4.5.1	The Controller Class . . . . .	36
4.5.2	Common DocumentHandler . . . . .	41
4.6	Well-formedness Components . . . . .	44
4.6.1	Control Component . . . . .	44

---

4.6.2	User Interface Component . . . . .	45
4.6.3	Functional Component . . . . .	46
4.7	Validation Components . . . . .	48
4.7.1	Control Components . . . . .	48
4.7.2	User Interface Component . . . . .	50
4.7.3	Functional Component . . . . .	51
4.8	XPath Components . . . . .	52
4.8.1	Control Component . . . . .	52
4.8.2	User Interface Component . . . . .	53
4.8.3	Functional Component . . . . .	53
4.9	DTD2Schema . . . . .	54
4.10	Implementation Plan . . . . .	55
<b>5</b>	<b>Implementation</b>	<b>57</b>
5.1	Common Components Implementaion Issues . . . . .	57
5.1.1	Controller . . . . .	57
5.1.2	DocumentHandler . . . . .	59
5.1.3	Error . . . . .	62
5.1.4	Status . . . . .	62
5.1.5	Message . . . . .	62
5.2	Well-formedness Plug-in Implementation Issues . . . . .	64
5.2.1	WellformednessController . . . . .	64
5.2.2	WellformednessDocumentHandler . . . . .	65
5.2.3	WellformedChecker . . . . .	65
5.2.4	StandardChecker . . . . .	66
5.2.5	WellFormedGUI . . . . .	66
5.3	Validation Component Implementation Issues . . . . .	68
5.3.1	ValidationController . . . . .	68
5.3.2	ValidationDocumentHandler . . . . .	69
5.3.3	Validator . . . . .	69
5.3.4	ValidationGUI . . . . .	70
5.3.5	ValidationErrorGUI . . . . .	70
5.3.6	ValidationError . . . . .	71
5.3.7	ValidationTest . . . . .	72

---

5.4	XPath Component Implementation Issues . . . . .	73
<b>6</b>	<b>Continuous Checking of Large Files</b>	<b>75</b>
6.1	Introduction . . . . .	75
6.1.1	The Challenge . . . . .	75
6.1.2	The Development Task . . . . .	75
6.1.3	Optimized Checking Strategy . . . . .	78
6.2	Design . . . . .	79
6.2.1	Data Structure for Storing Intervals . . . . .	79
6.2.2	Trimming the Data Structure . . . . .	85
6.2.3	The Optimized Checker . . . . .	86
6.3	Implementation . . . . .	91
<b>7</b>	<b>Testing</b>	<b>93</b>
7.1	Conditions . . . . .	93
7.2	Common Component . . . . .	93
7.3	Well-formedness Plug-in . . . . .	94
7.3.1	Well-formedness Control-Component . . . . .	94
7.3.2	Well-formedness User Interface . . . . .	94
7.3.3	Well-formedness Functional Component . . . . .	94
7.4	Validation Plug-in . . . . .	95
7.4.1	Validation User Interface . . . . .	95
7.4.2	Validation Functional Component . . . . .	95
7.4.3	Validation Control-Component . . . . .	95
7.5	Acceptance Test . . . . .	96
7.5.1	Acceptance Test of the Well-formedness Plug-in . . . . .	96
7.5.2	Acceptance Test of the Validation Plug-in . . . . .	97
7.5.3	Result of the Acceptance Tests . . . . .	98
<b>8</b>	<b>Further development</b>	<b>101</b>
8.1	Overall System . . . . .	101
8.1.1	Collection of Standard Messages in the Platform . . . . .	101
8.1.2	Redesign of the Overall System . . . . .	101
8.1.3	Better Integration with Under-laying OS . . . . .	102

---

8.2	Common Component . . . . .	102
8.3	Well-formedness Components . . . . .	102
8.4	Validation Components . . . . .	102
<b>9</b>	<b>Process</b>	<b>105</b>
9.1	Discussion of the OOAD Model . . . . .	105
9.1.1	Project Phases . . . . .	105
9.1.2	OOAD Tools . . . . .	105
9.2	Management Structure . . . . .	107
9.2.1	Protocols . . . . .	107
9.2.2	Management Roles . . . . .	110
9.3	Recommendations . . . . .	112
<b>10</b>	<b>Conclusion</b>	<b>113</b>
<b>A</b>	<b>System Requirements Specification</b>	<b>115</b>
A.1	Introduction . . . . .	115
A.1.1	Specification Overview . . . . .	115
A.1.2	Stakeholders . . . . .	115
A.1.3	References . . . . .	116
A.2	System Overview . . . . .	116
A.2.1	Definition . . . . .	116
A.2.2	Goals and Objectives . . . . .	116
A.2.3	AXE Context . . . . .	117
A.2.4	External Software . . . . .	118
A.3	Functional Requirements . . . . .	118
A.3.1	File-system . . . . .	119
A.3.2	Multiple Views of Files . . . . .	119
A.3.3	Version Control . . . . .	120
A.3.4	Searching in Files . . . . .	120
A.3.5	Validation . . . . .	120
A.3.6	Platform/Plug-in Structure . . . . .	120
A.3.7	User Defined Shortcuts . . . . .	120
A.3.8	Customization . . . . .	121

---

A.4	Quality Requirements . . . . .	121
A.4.1	Description of Quality Attributes. . . . .	121
A.5	Design Constraints . . . . .	124
A.6	Tests . . . . .	124
A.6.1	Unit Tests . . . . .	125
A.6.2	Acceptance Test . . . . .	125
<b>B</b>	<b>Protocols</b>	<b>127</b>
B.1	Code Standards . . . . .	127
B.2	CVS and Build Guidelines . . . . .	127
B.3	Interfaces and Design . . . . .	128
B.4	Meetings . . . . .	129
B.5	Project Team Collaboration Tool . . . . .	130
B.6	Testing . . . . .	131
<b>C</b>	<b>DTD</b>	<b>133</b>
<b>D</b>	<b>XML-Schema</b>	<b>135</b>
<b>E</b>	<b>testDTD.xml</b>	<b>137</b>
<b>F</b>	<b>testXSD.xml</b>	<b>139</b>

# List of Figures

3.1	Rich picture of the context of the Plug-in Collection. . . . .	9
3.2	The clusters of the Plug-in Collection. . . . .	10
3.3	The internal structure of the Common cluster. . . . .	11
3.4	The state chart for the Controller. . . . .	12
3.5	The statechart for the DocumentHandler. . . . .	13
3.6	The internal structure of the Well-formedness cluster. . . . .	17
3.7	The state-chart for the Well-formedness Checker. . . . .	18
3.8	The internal structure of the Validation cluster. . . . .	20
3.9	The state-chart for the Validator. . . . .	21
3.10	The structure of XPath. . . . .	23
3.11	The state-chart for the XPath. . . . .	24
3.12	The state-chart for the DTD2Schema. . . . .	26
4.1	The Principle behind the Message Queue. . . . .	30
4.2	Component architecture. . . . .	34
4.3	Class Diagram for Common. . . . .	36
4.4	Complete class diagram of <i>DocumentHandler</i> . . . . .	41
4.5	Class diagram for Status. . . . .	41
4.6	Class diagram for Message. . . . .	42
4.7	Class diagram for Error. . . . .	42
4.8	Complete class diagram of Controller . . . . .	43
4.9	The Well-formedness Checker classes. . . . .	44
4.10	Class diagram for WellformednessController. . . . .	45
4.11	Implementation class for WellformednessDocumentHandler. . . . .	45
4.12	Class diagram for WellFormedGUI. . . . .	46
4.13	The three status icons. . . . .	46

---

4.14	Class diagram for WellformedChecker. . . . .	47
4.15	Class diagram for StandardChecker. . . . .	47
4.16	Class diagram for the Validator. . . . .	48
4.17	Class diagram for ValidationController. . . . .	49
4.18	Class diagram for ValidationDocumentHandler. . . . .	49
4.19	The Validation GUI. . . . .	50
4.20	Class diagram for ValidationGUI. . . . .	50
4.21	Class diagram for ValidationErrorGUI. . . . .	51
4.22	Class diagram for ValidationError. . . . .	51
4.23	The Validate Class. . . . .	52
4.24	Class diagram for XPathController. . . . .	52
6.1	Illustration showing how XML can be modeled as a tree. . . . .	77
6.2	Illustration showing how XML sub-trees can be identified as intervals. . . . .	77
6.3	Visualization of the optimized checking strategy as cutting off tops in a structural view of an XML document. . . . .	79
6.4	An example of how the well-formed sub-trees of an XML document will be stored in the Interval Tree data structure. . . . .	81
6.5	An SDL diagram of the procedure for the optimized checker. . . . .	88
6.6	An SDL diagram of the procedure for handling changes in the document. . . . .	90
A.1	A rich picture of AXE. . . . .	117
A.2	The criterias of the design. Shows how the different quality attributes have been prioritized. . . . .	125

# Chapter 1

## Introduction

The Aalborg University XML Editor(**AXE**) is a collaborative application development project. Outlined, the project contains the following points of interest, which are explained in this chapter:

- The development process is done as a collaborative work.
- XML is the technological subject of the project.
- The task is to develop an application.
- Methods for application development is in focus.

### 1.1 Collaborative Work

The development task and process of this project is organized as a collaborative work (a so-called multi-project) of four project groups, where each group is assigned a specific part of the overall system.

The task of the four groups is to develop an advanced e**X**tensible Markup Language (XML) editor. The details of AXE and the subdivision of the development task are described in **Chapter 2 - Joint Analysis and Design** .

The development process, work, and result had been influenced by the multi-project setting and the fact that the system may be passed on to new students on the 5<sup>th</sup> Software Engineering semester. The multi-project setting will be discussed in **Chapter 9 - Process** . The chapter outlines the details of good and bad experiences gained in the multi-project.

## 1.2 XML

XML have had a major impact on data-formats and data-storage since it provides a common base enabling the definition, transmission, validation, and interpretation of data between applications.

The use of XML technology have spread to several different areas in computer science including several programming languages to data-base technologies.

## 1.3 The Task

The task assigned this group, is to provide a collection of XML processing tools for AXE. AXE is extendable through plug-ins, so the tools are designed as such. The Plug-in Collection features well-formedness checking and validation of XML documents, XPath querying and DTD to XML Schema conversion. Each of the features are described in-depth in **Chapter 3 - Analysis** and the chapters to follow.

A well-formedness plug-in providing instant error-reporting while editing large XML documents is developed. The plug-in is described in **Chapter 6 - Continuous Checking of Large Files** .

## 1.4 Development Method

The analysis and design phase is emphasized on this semester in relation to the semester topic “application development”. To support the system analysis and design course the model presented in “Object Oriented Analysis and Design”(OOAD) [1] are used. The course teaches a method for analyzing and designing a computer system in an organizational context. This method has intentionally been applied in this project to support the analysis and design phase.

However, the context of the Plug-in Collection is technical not organizational. For this reason it is considered difficult to apply a method for the project. Nevertheless the method is applied where it is found reasonable.

The resulting analysis document in **Chapter 3 - Analysis** is really more like a high-level design document. The low level design is described in detail in **Chapter 4 - Design** . The most analysis-like documentation is found in **Chapter 2 - Joint Analysis and Design** .

# Chapter 2

## Joint Analysis and Design

### 2.1 Interview

One of the premises of the multi-project was to base the requirements of the application on an interview with a person acting as customer. Because of that, an interview was made with that purpose in mind.

The result of this interview was, however, not exactly what the groups had expected. It did not match the project proposal presented at the start of the semester. Furthermore, the expected series of interviews with the customer was cut down to only this initial interview, so it appeared that the customer was not supposed to review the final program.

Therefore, it was decided to use the interview as inspiration from a potential user rather than a direct dictation of requirements from a customer. The system requirements specification (SRS) is thus the outcome of a discussion meeting held on basis of the interview. At this meeting the project managers from the groups made a list of requirements. This list formed the base of the SRS, which can be found in its full form in **Appendix A - System Requirements Specification** .

The reader is strongly encouraged to browse through the SRS document as it gives an overview of the overall system and puts this project into perspective. This SRS also contains some analysis of the overall system.

### 2.2 Delegation of Responsibility

As mentioned the groups should each have their own area of responsibility, and based on the SRS they were split into two areas. Both areas were occupied by two groups. The two areas are:

- Platform development
- XML plug-in development

It was chosen that the groups S501A and S502A should be in charge of developing the platform and S503A and S504A should supply it with XML plug-ins.

The responsibility of the groups are as follows. The areas are linked to their description in the SRS in **Appendix A - System Requirements Specification** :

- S501A:
  - **Section A.3.4 - Searching in Files**
  - **Section A.3.6 - Platform/Plug-in Structure**
  - **Section A.3.7 - User Defined Shortcuts**
  - **Section A.3.8 - Customization**

This group should make the general platform providing the plug-in structure, the general user interface of an editor, and the project structure.

- S502A:
  - **Section A.3.1 - File-system**
  - **Section A.3.3 - Version Control**

This group implements the file-system and the version control system.

- S503A:
  - **Section A.3.2 - Multiple Views of Files**

This group should provide an XML editor which handles large XML files, color-coding, auto-completion etc.

- S504A (which is the group writing this report):
  - **Section A.3.5 - Validation**

This group should handle well-formedness checking and validation of XML documents.

# Chapter 3

## Analysis

### 3.1 The Task

This chapter outlines what the overall aim for the project is. It gives the analysis of the the Plug-in Collection and a description of the different XML technologies involved.

#### 3.1.1 Purpose

The purpose of the Plug-in Collection is to equip AXE with the features of checking XML documents for well-formedness, validity and querying XML documents with XPath.

Well-formedness and validity of XML documents together with the XPath standard are described in the World Wide Web Consortium's (W3C) recommendations. The following is an quick overview of the three terms. A detailed description of XML can be found in [2]. The following paragraphs gives a description of the involved W3C standards.

**Well-formedness** The W3C defines a textual object as a well-formed XML document if:

- It has one and only one root element.
- All opening tags are matched by their respective closing tags.
- All elements are properly nested.

The following shows an example of a well-formed XML document:

```
<root>
  <element1>
    <element2>
    </element2>
  </element1>
</root>
```

This example is well-formed because it fulfills the above stated requirements: There is one and only one root tag, which is `root`. Each opening tag has a corresponding closing tag: `<root>` matches `</root>`, `<element1>` matches `</element1>` and `<element2>` matches `</element2>`. And the tags are properly nested: `element2` is opened after and closed before `element1` is closed, and `element1` is opened after and closed before `root`.

The following shows an example, that is not well-formed.

```
<root>
  <element1>
    <element2>
  </element1>
  </element2>
</root>
```

It fulfills the first two properties of well-formedness, but not the last one about nesting: `element2` is opened after `element1` is opened, but `element1` is closed before `element2` is closed. Thus, they overlap, which is not allowed. A detailed description is found on [2].

**Validity** W3C defines an XML document as valid if it has one or more associated Document Type Definitions (DTDs) or Schemas and complies with the constraints described therein.

A DTD defines the syntax and structures of an XML document. It describes which elements can go where and which attributes and data types are allowed. A DTD can also be written as an XML Schema, an example of a DTD see **Appendix C - DTD** for a schema example see **Appendix D - XML-Schema** . The main difference is that the XML Schema is itself an XML document and it is more specific about data types of attributes than a DTD.

Well-formed XML documents have a structure that can be interpreted as trees. DTDs define a tree-structure in a certain pattern or form and has certain properties. To validate an XML document you derive its tree-structure and see if it fits into the structure defined by the DTD. If it fits, the XML document is valid. If not, its invalid. For a detailed description see [2].

**XPath** XPath is a standard defined by W3C for addressing parts of an XML document. The basic concept of XPath is similar to paths on a file system. Both XML and file systems can be viewed as trees. The path to a file could be:

```
/disk/folder/subfolder/file.txt
```

XPath addresses elements similarly:

```
/root/element/subelement/subsubelement
```

Apart from this simple way of addressing elements, W3C defines some advanced addressing techniques that are also a part of the XPath standard [3].

**DTD2Schema** DTDs are used to specify the syntax and structure of XML documents. A problem with DTDs is that they do not specify types on attributes, which can lead to inconsistent data being accepted. This can be avoided by using XML Schema, which is why a DTD2Schema converter comes in handy. XML Schema is a W3C standard defined in [4], [5] and [6]. XML Schemas are written as XML, an example can be seen in **Appendix D - XML-Schema** .

### 3.1.2 System Definition

#### FACTOR

- **Functionality:** A technical set of tools designed as plug-ins expanding the functionality of AXE.
- **Application domain:** AXE.
- **Conditions:** The Plug-in Collection is a set of plug-ins to AXE. The plug-ins can be loaded and unloaded. The Plug-in Collection should be documented thoroughly to make it maintainable and reusable. Knowledge of algorithmic problem solving techniques and existing components should be used when applicable.
- **Technology:** The Plug-in Collection must be runnable on Microsoft .NET's Framework and use relevant XML technologies.
- **Objects:** Plug-ins, XML documents, DTDs and XML Schemas, interfaces to AXE and AXE itself.
- **Responsibility:** Check for well-formedness and validity of XML documents. Provide functionality to search XML-documents. Converting DTDs to XML-Schemas.

These criterias can be summarized into the following system definition.

The Plug-in Collection should be designed as a number of plug-ins to AXE. It should be possible to load the plug-ins on demand or automatically triggered by relevant file types (.xml, .xhtml .svg etc.). The Plug-in Collection must provide functionality to check for well-formedness and validity of XML documents as well as querying the XML documents with XPath and converting DTDs to XML Schemas.

The validation plug-in must be able to validate up against DTD and XML Schema. The result and status of the well-formedness check and validation should be shown through the plug-in's associated user interface. The well-formedness check should be run continuously, always showing the current well-formedness status of the active document. Validation should be run on demand.

Functionality for converting DTDs to XML Schemas should be provided to help migration from the one format to the other. The Plug-in Collection should be well documented to increase maintainability to in help future development.

Solutions to problems should use algorithms, problem solving techniques and third-party components to the extend possible.

### 3.1.3 Context

The Plug-in Collection being plug-ins to AXE is fully dependant on AXE for all external resources and functionality. This involves file and internet access, user input and system output. The rich picture shown in **Figure 3.1** illustrates this. The Plug-in Collection and AXE are boxed together to indicate that in practice these melt together so that the user will see them as one overall system. System output to the user terminal – including graphical user interface (GUI) – illustrated by the “OK” sign is also connected to this boxed area.

The application domain of the Plug-in Collection is XML documents, and it should be able to handle multiple documents simultaneously. XML being a W3C technology, many XML resources like DTDs and XML Schemas are located on the Web. As a plug-in, the Plug-in Collection is not allowed nor able to access these resources directly, and thus, AXE has to make this functionality available to the Plug-in Collection, which is depicted as the The Internet in **Figure 3.1**.

Also, the Plug-in Collection needs access to the local file system. The ability to read files is obviously required, which is indicated by the arrows going from the DTD and meta-data documents through “Local” and towards AXE in **Figure 3.1**. The arrows going the opposite way indicates the need of the Plug-in Collection to be able to write. First of all, the DTD to Schema Converter should be able to save the result of the conversion. Secondly, the Validator and the Well-formedness Checker will gather and maintain different kinds of meta-data while processing an XML document, and these data should be written to disk when the document is closed, and reused the next time the file is opened.

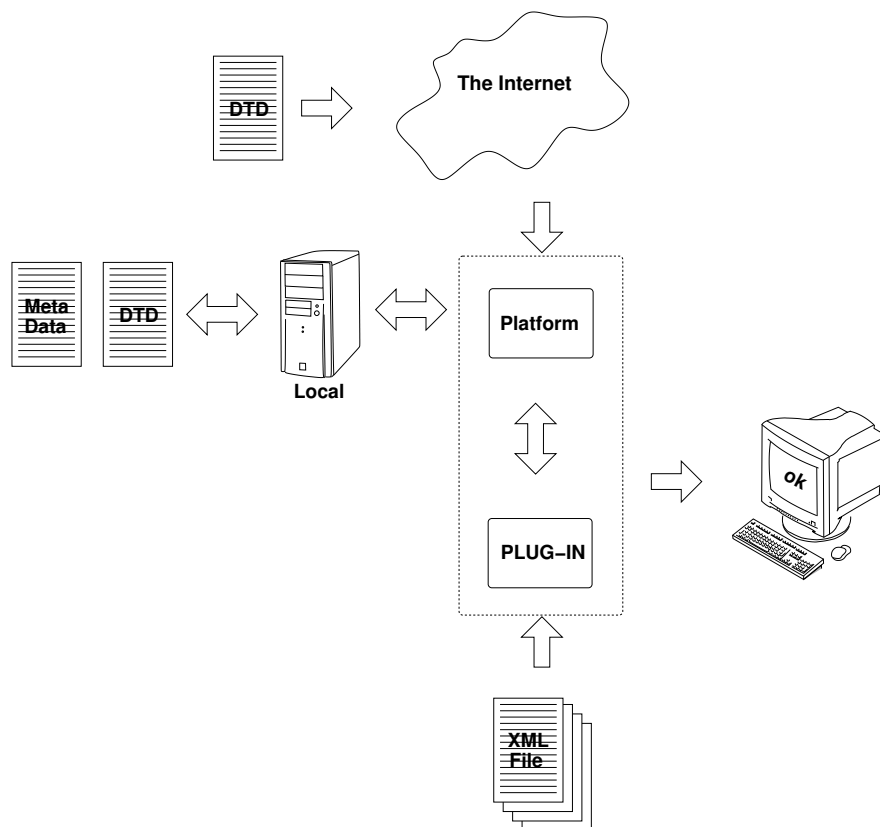


Figure 3.1: Rich picture of the context of the Plug-in Collection.

## 3.2 Cluster Structure of the the Plug-in Collection

As seen in **Figure 3.2** and as it is indicated in the **Section 3.1.2 - System Definition**, the Plug-in Collection is made up of five clusters: Common, Well-formedness, Validation, XPath and DTD2Schema. These are described in greater detail in the coming sections.

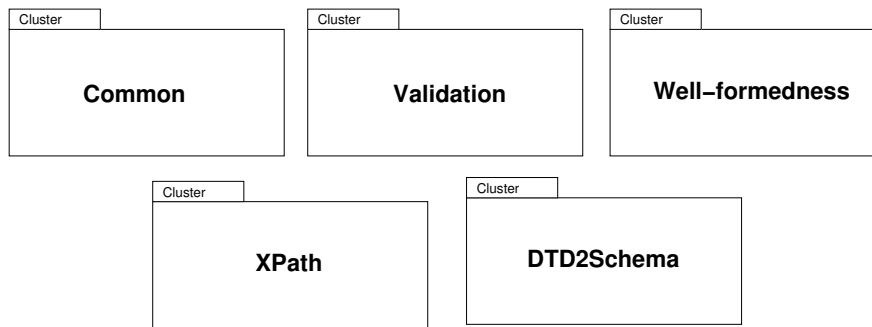


Figure 3.2: The clusters of the Plug-in Collection.

## 3.3 Common

This section describe the structure, behavior, usage, and functions of the common cluster. This cluster will provide all plug-ins with a common non changing interface to the platform.

### 3.3.1 Structure

The Common cluster provides encapsulation of the communication between the actual plug-ins and AXE. If the communication interface to AXE changes, then only the set of classes in Common has to be updated to reflect these changes. Furthermore, all shared functionality of the different plug-ins are put into the Common cluster.

Internally, Common consists of four classes: *Controller*, *DocumentHandler*, *Error* and *Status*. Their relationship is depicted in **Figure 3.3**. As can be seen, all the relationships are associations. The base class is the *Controller*, which interfaces AXE and controls the behaviour of the actual plug-in. As the plug-ins should be able to handle multiple XML documents, the *Controller* can have multiple *DocumentHandlers* attached. Each *DocumentHandler* should internally represent and handle a single document opened in AXE. Each document should have a status and can have multiple errors, which explains the associated *Status* and *Error* classes.

The *Controller* and the *DocumentHandler* classes written in italic are abstract, which indicates that they only contain the basic, shared functionality that has to be applied from the actual plug-ins to be usable.

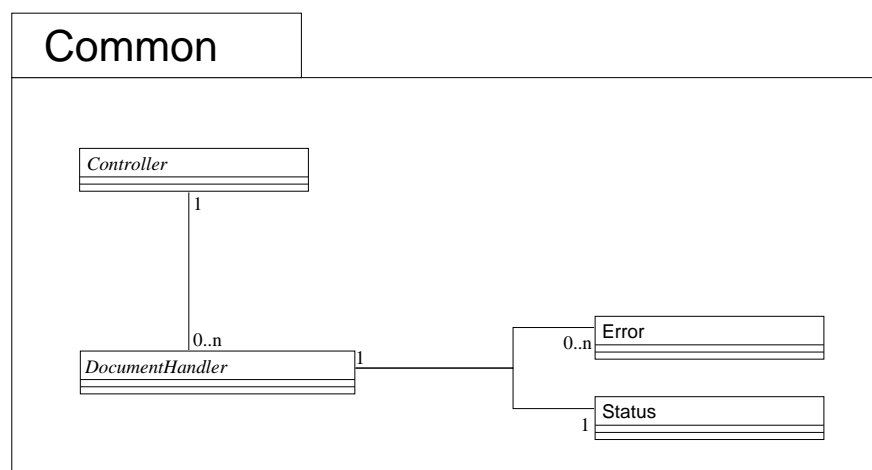


Figure 3.3: The internal structure of the Common cluster.

### 3.3.2 Behavior

#### Statechart

The Controller class should handle all communication with AXE including manipulation of the GUI within AXE. The communication is two-way:

- AXE loads the plug-ins and it provides information about changes in AXE and in other plug-ins.
- The plug-in retrieves resources from AXE and reports back status or errors it has come up with.

The Controller function as a switchboard by maintaining a list of all the currently opened files in AXE. As mentioned, it has a `DocumentHandler` for each file. The file shown in the editor's main window is called the active file and receives special attention. The Controller should make sure that the plug-in primarily works on the active file.

The state-chart in **Figure 3.4** shows how the Controller acts on different events. Before any work can be done by the plug-ins, the Controller class must be loaded which the event *Load* indicates. When the Controller is loaded it enters the *Sleep* state. In the *Sleep* state, the Controller class waits for incoming messages from the message queue, and it is ready for any outgoing messages that needs to be posted to the message queue. When this happens, the Controller enters the *Active* state. When the incoming message has been processed or when the outgoing message gives an *Message sent* event, then the Controller again enters *Sleep*.

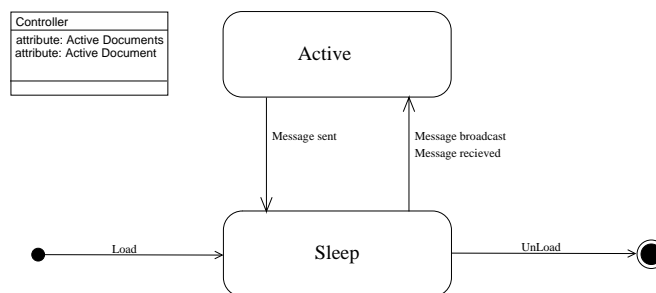


Figure 3.4: The state chart for the Controller.

The state-chart in **Figure 3.5** shows how the `DocumentHandler` reacts on different events. As it is described above, the `DocumentHandler` acts as a leash to a file. All events concerning the `DocumentHandler` has to do with operations on files. The two events that the `DocumentHandler` responds to are *File Opened* and *File Closed*. The class is in the state *Active* from the moment the associated file is opened and until it is closed.

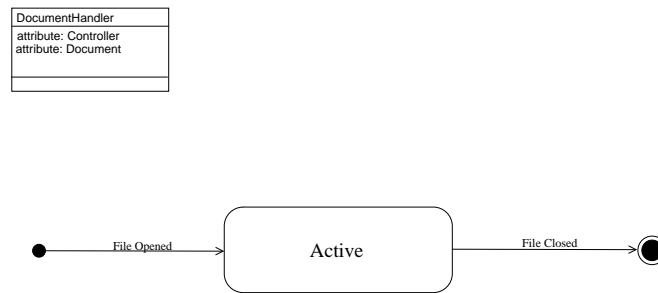


Figure 3.5: The statechart for the DocumentHandler.

### 3.3.3 Usage

#### Actor Specification

The tables, **Table 3.1**, **Table 3.2**, **Table 3.3** and **Table 3.4** shows the four actors that the the Plug-in Collection interacts with.

User
<p><b>Purpose:</b> A person that uses the system (AXE and the Plug-in Collection) to edit XML documents and check well-formedness and validity.</p>
<p><b>Example:</b> Uses the a menus to select a functionality that will initiate checks of well-formedness or validation.</p>

Table 3.1: User Actor Definition.

<b>Platform</b>
<b>Purpose:</b> A system responsible for communication and managing file access.
<b>Characteristics:</b> Handles all communication with the underlying systems. It is also responsible for the loading of the the Plug-in Collection. The Platform can handle many simultaneous open XML-documents.
<b>Example:</b> When a file is opened the Platform sends a message on its message queue.

Table 3.2: Platform Actor Definition.

<b>Editor plug-in</b>
<b>Purpose:</b> A system that make changes in an XML-document.

Table 3.3: Editor plug-in Actor.

<b>Other plug-ins</b>
<b>Purpose:</b> No generally defined purpose.
<b>Characteristics:</b> Any plug-in that some time in the future may want to react to messages posted by any plug-in in the Plug-in Collection.

Table 3.4: Future plug-in.

## Use-cases

The following use-cases is identified.

---

### *File Opened*

**Pattern:** File Opened is initiated by the user and is passed on through the message queue. The different plug-ins receive the “file opened” message and take the appropriate actions.

---



---

### *File Closed*

**Pattern:** File Closed is initiated by the user and is passed on through the message queue. The different plug-ins receive the “file closed” message and take the appropriate actions.

---



---

### *File Changed*

**Pattern:** File Changed is initiated by the user and is passed on through the message queue. The different plug-ins receive the “file changed” message and take the appropriate actions.

---



---

### *Active Document Changed*

**Pattern:** Active Document Changed is initiated by the user. A change occurs in a document if the file has been edited.

---

## 3.3.4 Functions

**Table 3.5** shows candidates to functions that may be shared among the different plug-ins.

Function	Complexity
<i>Load and Setup a Plug-in</i>	Medium
<i>UnLoad a Plug-in</i>	Medium
<i>Send Message to Platforms Message queue</i>	Medium
<i>Receive Message from Platforms Message queue</i>	Medium
<i>Handle Communication with the Platforms Managers</i>	Medium

Table 3.5: Function Listing.

### **Function Description**

***Load and Setup a Plug-in*** This function takes care of the initialization of a plug-in when it is loaded. It reads the configuration and sets up the plug-in in the context in which it must work. Similar to the constructor of a class.

***Unload a Plug-in*** Procedures taken when the plug-in is unloaded. Similar to the destructor or finalizer of a class.

***Send Message to Platforms Message Queue*** This function provides a plug-in with the ability to send messages to the platforms message queue. Sending messages to the message queue is the principal way that a plug-in can communicate with another plug-in.

***Receive Message from Platform Message Queue*** Provides a plug-in with the ability to receive messages sent by another plug-in or by the platform.

***Handle Communication with the Platforms Managers*** Provides an interface with communication to the platform's managers like GUI and menus.

## 3.4 Well-formedness

This section describe the structure, behavior, usage, and functions of the well-formedness cluster.

### 3.4.1 Structure

The Well-formedness cluster represents the first of the four actual plug-ins, and its structure is shown in **Figure 3.6**. It has a controller and a number of document handlers. Each document handler manage one XML-document. To each document handler there is an associate Well-formedness Checker class. The Well-formedness Checker class is to encapsulate all the functionality needed to check an XML-document for well-formedness.

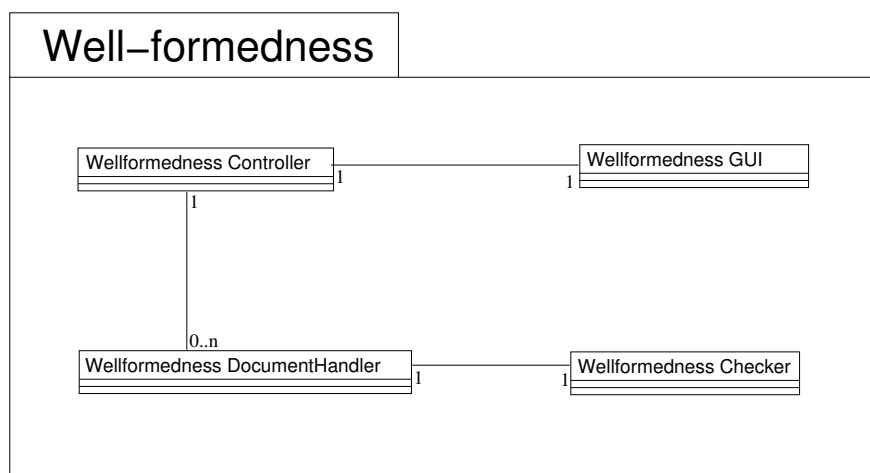


Figure 3.6: The internal structure of the Well-formedness cluster.

### 3.4.2 Behavior

#### State-chart

The **Figure 3.7** show the state-chart for the Well-formedness Checker. The Well-formedness Checker plug-in starts its checking on event *File Opened*. When the file is opened the plug-in checks for status for the meta-data on event *Checking meta-data*. If the meta-data is ok indicated by the event *ok* the Well-formedness Checker enters into the state *Sleep* where it waits for the event *XML-document changed*. On the event *XML-document changed* the Well-formedness Checker (again) checks the XML-document for well-formedness. If either of the events *Complete* or *Error* is encountered the Well-formedness Checker-specific goes back to the state *Sleep*. The event *Complete* corresponds to that the Well-formedness Checker do not encounter

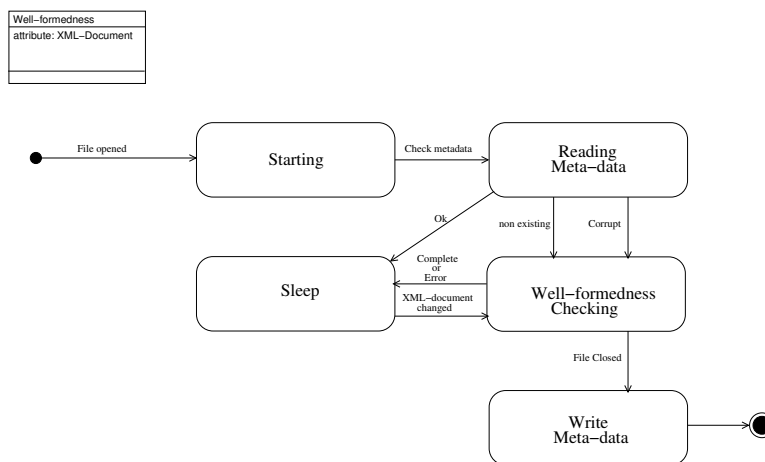


Figure 3.7: The state-chart for the Well-formedness Checker.

any errors during its check. Vice versa for the event *Error*. If the file associated with the Well-formedness Checker is closed the Well-formedness Checker writes meta-data to storage.

### 3.4.3 Usage

#### *Start Well-formedness Checker*

**Pattern:** Start Well-formedness Checker is initiated by the “file opened” and “file changed” messages from the message queue sent by the AXE and other plug-ins. If the Well-formedness Checker is already running, then the process is restarted from the beginning of the document.

### 3.4.4 Functions

**Table 3.6** shows candidates to functions that are specific for the Well-formedness Checker. It shares the functions located in Common, see **Section 3.3.4 - Functions**.

Function	Complexity
<i>Start the Well-formedness Checker</i>	Simple
<i>Check for well-formedness</i>	Complex

Table 3.6: Function listing for the Well-formedness Checker.

### Function Description

***Start the Well-formedness Checker*** This function enables one the user or AXE to start checking an XML-document for well-formedness.

***Check for well-formedness*** Provides the Well-formedness Checker with functionality to check an XML document for well-formedness. For a more detailed discussion of the well-formedness checker see **Section 4.6.3 - Functional Component**

### 3.4.5 User Interface

The user interface of the Well-formedness Checker should not entail more than a set of icons. The icons should display the outcome of the Well-formedness Checker, which is the well-formedness status of the active document. The outcomes are one of the following:

- Well-formed
- Not Well-formed
- Unknown

## 3.5 Validation

This section describe the structure, behavior, usage, and functions of the validation cluster.

### 3.5.1 Structure

The Validation cluster represents the second of the four actual plug-ins. It encapsulates all functionality that is required to enable AXE to validate XML documents. As you can see in **Figure 3.8**, it is similar to Well-formedness in **Figure 3.6** and should explain itself if you switch the labels “well-formedness” and “validation”. The functionality in Validator is, of cause, different from Well-formedness Checker in that it handles the process of validating XML documents.

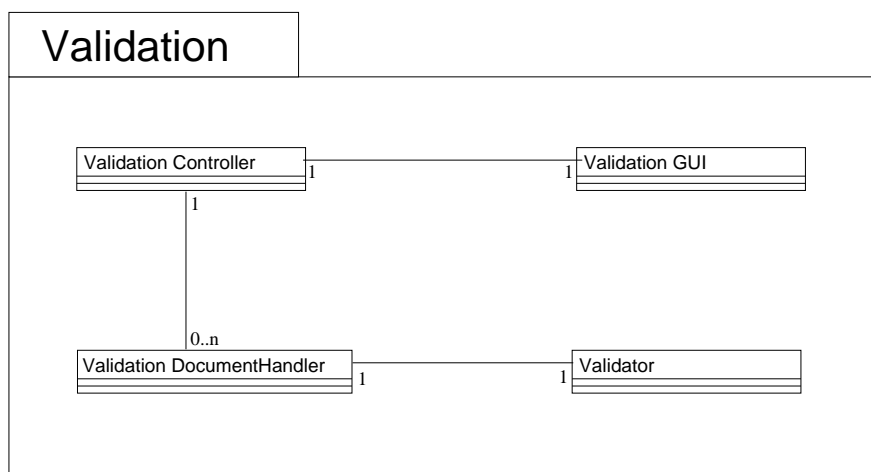


Figure 3.8: The internal structure of the Validation cluster.

### 3.5.2 Behavior

#### State-chart

The state-chart for the Validator is seen in **Figure 3.9**. It is decided that it should only be run on demand by the user. The user supply the event *validate* to the Validator. The Validator then enters the state *Starting*. It then loads any meta-data for the active XML document. If the meta-data reveals that the document already has been checked and found valid, then the valid message is presented. If the meta-data is either non-existing or corrupt the events *Non-existing* or *Corrupt* occurs and the Validator checks the validity of the active XML document. When the Validator finishes either the event *Valid* or *Not-Valid* occurs. Depending on the event either a list of encountered errors or a message saying that the active XML document is valid is displayed. File handling

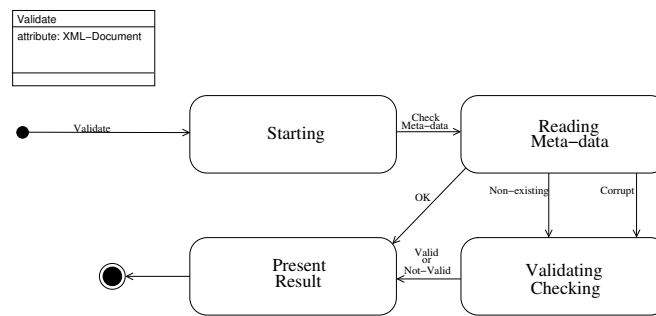


Figure 3.9: The state-chart for the Validator.

is done by the class `Validation DocumentHandler` and validation functionality is in the class `Validator`.

### 3.5.3 Usage

---

#### *Start Validation*

**Pattern:** Start Validation is initiated by either a user, AXE or a future plug-in. The actor starts the validation process by selecting a menu in the main editor window. The Validator plug-in can be configured to start when a “file opened”-message is posted on the message queue. Any meta-data is read, and appropriate action is taken:

- If the meta-data exists and is up-to-date the validation process is not started.
  - If the meta-data exists but is not up-to-date the validation process is started.
  - If the meta-data does not exist the validation process is started.
-

Function	Complexity
<i>Start the Validator</i>	Simple
<i>Check for Validity</i>	Complex

Table 3.7: Function Listing for Validator.

### 3.5.4 Functions

**Table 3.7** shows a list of candidates to functions that must be provided beside the functions from Common.

***Start the Validator*** This function is meant to provide an actor with the ability to start the Validator.

***Check for Validity*** This function checks if an XML document is valid or reports any errors found.

### 3.5.5 User Interface

The Validator 's user interface supplies the user of AXE with an overview of which validation errors were encountered during a check for validity. The user interface must provide the ability to go from the display of an error to the actual position of the error in the XML document.

The user interface of the validation plug-in consists of two parts. A menu for initiating the validation process, and a graphical component to show errors in the document. This component should be placed in the side-pane of AXE in order to allow viewing both errors and the XML document at the same time. The user should have the opportunity to get each error marked in other plug-ins by clicking a button associated with each error.

## 3.6 XPath

This section describe the structure, behavior, usage, and functions of the xpath cluster.

### 3.6.1 Structure

The structure of the XPath is shown in **Figure 3.10**.

Each XPath DocumentHandler represents an open XML document. The multiplicity between the XPath Controller and the XPath DocumentHandler represent that one controller can manage many opened files. To handle the actual query on an open document, each document handler has an associated class. The XPath class that is associated with the XPath DocumentHandler and encapsulates the functionality needed to make the query.

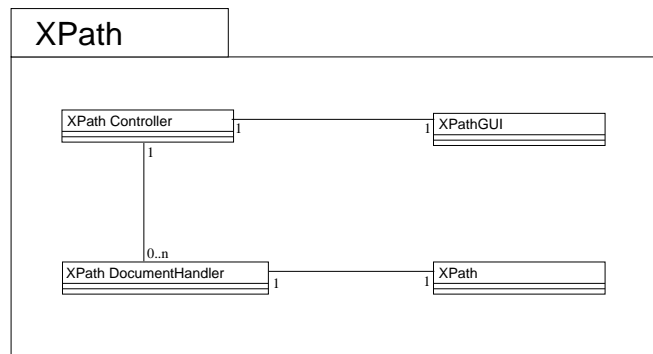


Figure 3.10: The structure of XPath.

### 3.6.2 Behavior

#### State-chart

The state-chart for the XPath plug-in is shown in **Figure 3.11**. The XPath plug-in is only run on user demand. The user supply the plug-in with the event *query* by requesting a query to be performed on the active XML-document. From the state Starting the XPath plug-in checks to see if an indexing structure for the active XML document already exists. If the indexing structure does not exist the event *not found* occurs. The plug-in will then enter into the state Build Indexing structure and try to build the structure. If the indexing structure does exist the event *found* occurs and the plug-in enters into the state Querying, where it performs the requested query on the XML document. The plug-in enters into the state Present Search result when the query has been performed and the result of the query is shown. When the event *File Closed* occurs the plug-in saves its current indexing structure and exits. If the plug-in can not build the indexing structure the plug-in exits.

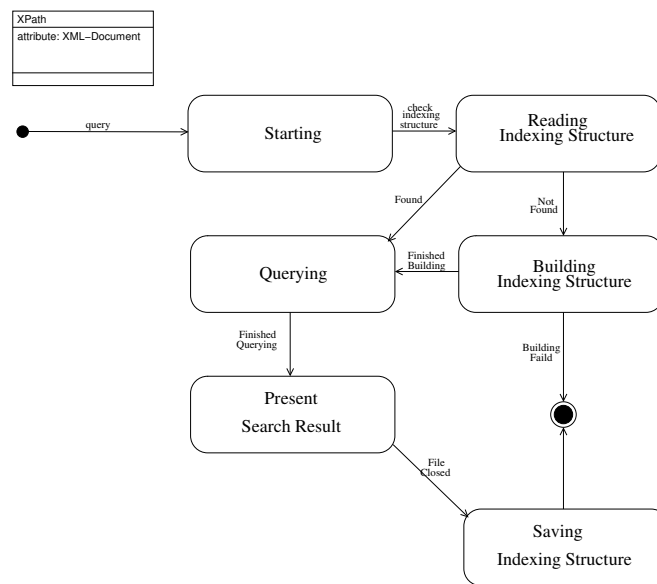


Figure 3.11: The state-chart for the XPath.

### 3.6.3 Usage

<i>Query</i>	
<b>Pattern:</b> The query is started by user and searches the build index for the query.	

### 3.6.4 Functions

Table 3.8 shows the different functions in the XPath cluster.

Function	Complexity
<i>Build index</i>	Complex
<i>Query</i>	Simple

Table 3.8: Function Listing for XPath.

#### Function Description

*Build index* This function build the index structure from the XML document to be searched in.

*Query* Queries the structure,

### **3.6.5 User Interface**

The user interface of the XPath cluster should consist of a text field to type the query and a place to display the results.

## 3.7 DTD2Schema

This section describe the structure, behavior, usage, and functions of the DTD2Schema cluster.

### 3.7.1 Behavior

#### State-chart

The state-chart for the DTD2Schema plug-in is shown in **Figure 3.12**. The DTD2Schema plug-in is only run on user request. The DTD2Schema runs when the event *convert* occurs resulting in the DTD2Schema plug-in entering the state Loading. In the Loading state the plug-in tries to load the DTD to be converted. If the loading fails the plug-in exits. Loading success is indicated by the event *OK* and the plug-in enters the Converting state where the DTD is being converted to an XML-schema. If the conversion succeeds the result is shown to the user in the Present Result state. If the converting fails the plug-in exits. If the user is satisfied with the conversion the event *result OK* takes the plug-in into the state Saving where the produced XML Schema is saved to disk. If the user wants to edit the resulting Schema the plug-in enters the state Edit where the looping *changing* event represents changes to the produced XML Schema document. When the user is satisfied with the changes the event *Done* makes the plug-in enter the state Saving. When the produced XML Schema document is saved and the plug-in exits.

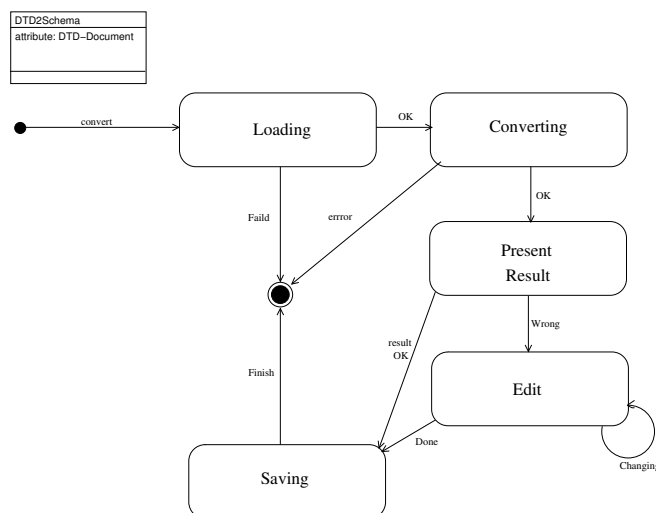


Figure 3.12: The state-chart for the DTD2Schema.

### 3.7.2 Usage

The DTD2Schema plug-in is intended as a tool supporting the migration from DTD to schema based XML formats.

### 3.7.3 Functions

On **Table 3.9** the functions of the DTD2Schema can be seen.

Function	Complexity
<i>Open a DTD</i>	Simple, using common dialogs
<i>Convert DTD to Schema</i>	Complex
<i>Allowing user to specify datatypes</i>	Complex
<i>Save the DTD</i>	Simple, using common dialogs

Table 3.9: Function Listing for DTD2Schema.

### 3.7.4 User Interface

The user interface of the DTD2Schema plug-in should allow the user to:

- Open the DTD file to convert.
- Specify data type constraints.
- Save the resulting.



# Chapter 4

## Design

### 4.1 Corrections and Assumptions to the Analysis

Corrections and assumptions made to the analysis are summarized in the next two sections. The correction to the analysis comes from changes in the way AXE is accessed. The assumptions listed are on what events will cause AXE to send a message.

#### 4.1.1 Corrections

During the analysis phase the way that plug-ins communicate with AXE changed. AXE now implements the communication with a message manager, the “Message Queue”. **Figure 4.1** shows the principles behind AXE plug-in communication.

In order for a plug-in to be considered a true AXE plug-in must implement the IPlugin interface.

#### 4.1.2 Assumptions

As **Figure 3.1** imply there is some communication between a plug-in and AXE. The communication will be in the form of messages. The messages will be sent on events that occur in AXE. The following events are assumed to make AXE send a message.

File Open

File Close

File Change

The communication works by letting plug-ins subscribe to events that are relevant to them.

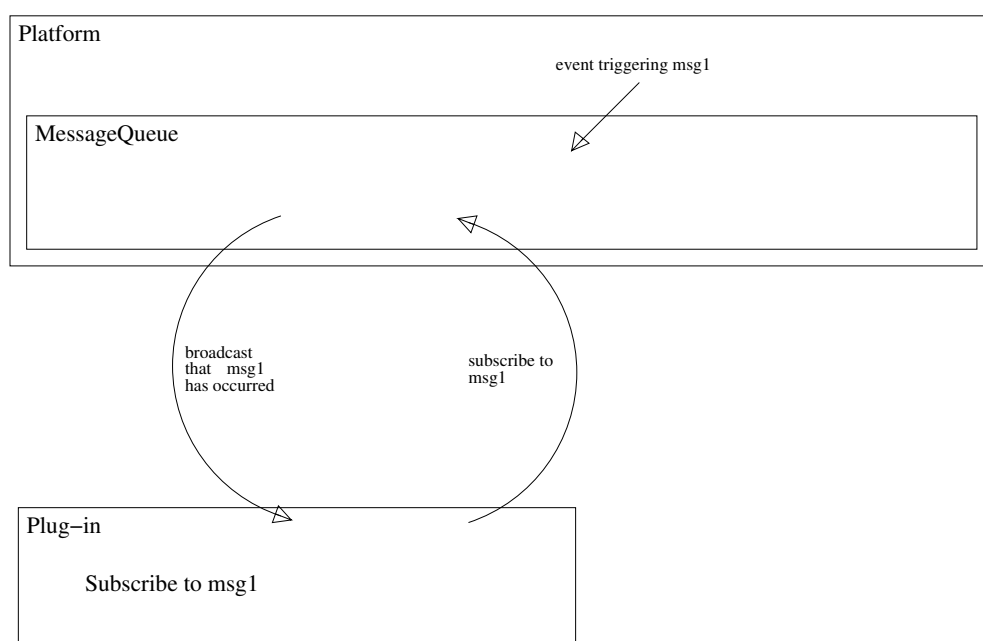


Figure 4.1: The Principle behind the Message Queue.

## 4.2 Quality Attributes

Knowing that the project is to be handed over to others has influenced how quality attributes are prioritized. The prioritizing of quality attributes is based on the group's own experience and understanding of the difficulties that are associated with a project being handed over. The quality attributes is weighted as it is shown in **Table 4.1**

### Discussion of Individual Quality Attributes

Quality attributes that have a different priority or otherwise differs from the quality attributes setup in the system requirements specification in **Appendix A - System Requirements Specification**, are discussed. The quality attributes in the system requirements specification are taken from the Open Project Framework(OPF) [7]. The quality attributes listed in **Table 4.1** are taken from the OOAD book, see [1]. Where there is a discrepancy between OPF and OOAD the mapping relationship is given. The discrepancies are discussed in the coming paragraph.

**Correct** Changed from "Important" to "Very Important". Either the Well-formedness Checker and the Validator should return the actual status of the XML document checked, or else it should fail. If an incorrect status is ever given the functionality is useless.

**Understandable** This has no direct mapping to any quality attribute in OPF. It is rated as "Important" as it will help decreasing the amount of time that coming groups

will have to spend understanding of the system.

**Intergratable** Changed from “Irrelevant” to “Less Important”. It is rated as “less important” and not “Very Important” although our task is to design a set of plug-ins. This decision is made since our set of plug-ins by nature is integratable. The intergratable quality attribute is therefor to some extend trivially obtained.

Criterion	Very Important	Important	Less Important	Irrelevant	Trivial Applicable
Usable		x			
Secure				x	
Efficient		x			
Correct	x				
Reliable				x	
Maintainable	x				
Testable	x				
Flexible			x		
Understandable		x			
Reusable		x			
Portable					x
Intergratable			x		

Table 4.1: Prioritized Design Criteria

## 4.3 The Technical Surroundings

This section gives a short description of the equipment, programs and systems that are used in creating the Plug-in Collection.

### 4.3.1 Equipment

AXE and the Plug-in Collection is to run on a standard PC equipped with a reasonable amount of RAM and disc space. What is considered reasonable is implicitly specified in the system requirements specification, see **Appendix A - System Requirements Specification** . Other requirements to the equipment comes from the programs needed see the following section.

### 4.3.2 Programs Needed

All cooperating groups decided on using Microsoft Visual Studio and one of the Microsoft .NET Framework supported programming languages. The choice was the C# programming language.

The decision to use the Microsoft .NET Framework put some additional requirements on the equipment on which the system must be able to run. The equipment needs to support all the requirement that the Microsoft .NET Framework has.

To develop AXE and the Plug-in Collection it has been decided that the same programming language should be used by all participating groups. The decision to use the same programming language is to insure that all groups with relative ease is able to read and understand the source code produced by the other participating groups.

## 4.4 Architecture

The Plug-in Collection consists of four plug-ins. The four plug-ins are made for the AXE platform. Since the plug-ins are made for the same platform all the plug-ins implement the same interface. Therefore they will all share the same basic component architecture described in the coming section.

### 4.4.1 Component Architecture

Intuitively, the architecture of the Plug-in Collection is a plug-in architecture. **Figure 4.2** illustrates the components of a single plug-in. The architecture for the plug-in consists of three components, which are the Control, the User Interface, and the Functional components. The following section will explain the components of the architecture.

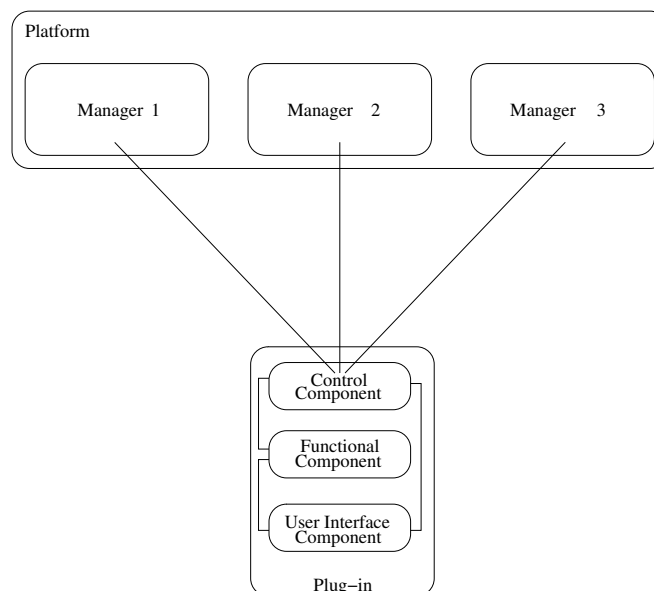


Figure 4.2: Component architecture.

### **Description of the Components**

**Control Component** This component is responsible for communication to the platform managers. Among the functionality provided by the managers is functionality to manipulate GUI. The user interface and functional components make use of the methods provided by the Control components to show results to a user.

**User Interface Component** This component is in charge of making GUI and updating the GUI with relevant information. The information comes from the functional component.

**Functional Component** This component contains the actual working classes of the plug-ins. It relies on the control component for access to the resources it needs.

## 4.5 Common Component

This section discusses the classes and methods common for all the developed plug-ins developed. It gives a short description for each of the methods and classes involved.

### 4.5.1 The Controller Class

The functionality that this class provides is as stated in **Section 3.3 - Common** communication with AXE. The class-diagram for the class is seen in **Figure 4.3** The *Controller* achieves its functionality – platform to plug-in communication – by wrapping all method calls to the managers of AXE. The encapsulation provides the the Plug-in Collection with a non-changing interface to AXE. The decision to encapsulation all communication is to make the the Plug-in Collection more robust.

The methods of the *Controller* can be divided into three main categories. **Table 4.2** list the categories and gives a short description of each. In **Figure 4.8** a complete overview of the *Controller* is given.

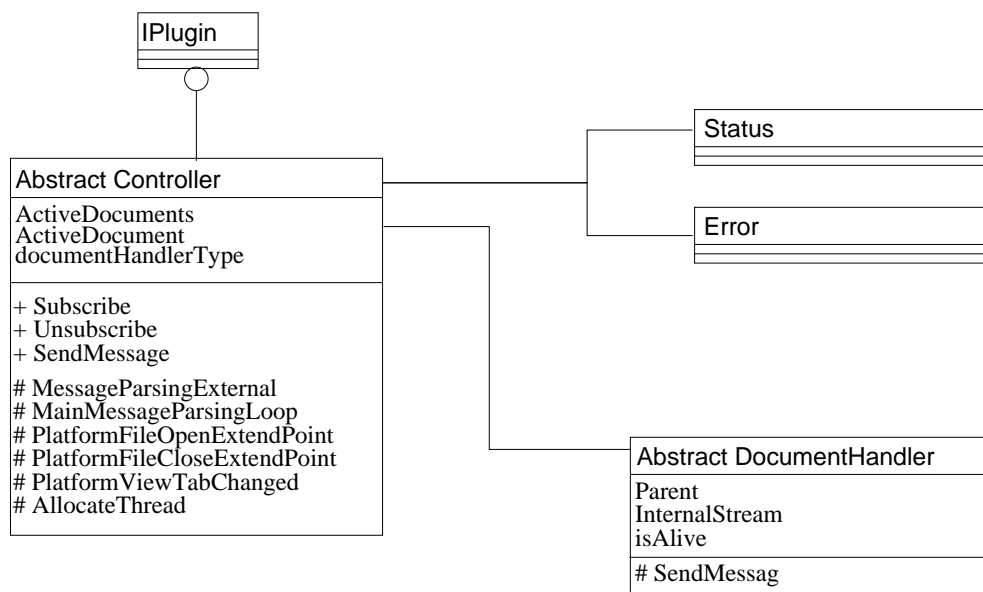


Figure 4.3: Class Diagram for Common.

### The Controller's Manager Communication Methods

The methods that communicate with the managers of AXE can be further subdivided into two parts. One part that encapsulates the communication with managers that are responsible for the GUI. The other part encapsulate communication with the underlying file system and message communication. In the following paragraphs a detailed description of the two parts are given.

Categories	Description
Manager Communication Methods	Methods that encapsulate communication with AXE
IPlugin Interface methods	Method that ensure that the plug-ins can be recognized as AXE plug-ins
Auxiliary Methods	Methods that plays an auxiliary role to either of the above method categories

Table 4.2: The Method Categories of the abstract *Controller* class

**Part One** The methods that setup and add components to the GUI of AXE.

***AddToolBarButton:*** Adds a button to the toolbar in AXE.

**Augments:** icon:Image

***AddIconPanel:*** Adds a icon on the status-bar in AXE.

**Augments:** icon:Icon, tooltipText:string

***AddProgressBar:*** Adds a progress-bar on the status-bar in AXE.

**Augments:** taskName:string, numberOfSteps:int

**Returns:** AProgressBarTask

***UpdateProgressBar:*** Updates the progress-bar with a new position.

**Augments:** AProgressTask

***AddSidePane:*** Adds a side pane in AXE.

**Augments:** Pane

***MenuItem:*** Adds a menu to AXE.

**Augments:** menuText

**Returns:** AMenuItem

***AddSubMenuitem:*** Adds a sub-menu to a menu in AXE.

**Augments:** aMenuItem:AMenuItem, menuText:string

***PostMessageOnStatusBar:*** Sends a string to be posted on the status-bar on AXE.

**Augments:** message:string

***UpdateGUI:*** Updates GUI elements on AXE.

**Augments:** status:object, error:object

**Part Two** To communicate with the message queue the method `SendMessage` is provided. As the interface to the file system is not in a stable state. The methods to access the file system are not considered.

***SendMessage:*** Sends a message on the message-queue.

**Augments:** `message:Message`

### The Controller's IPlugin Interface Methods

The interface IPlugin provide a number of different methods to help identify the plug-in.

**Subscribe:** Lets AXE know that the plug-in is interested in type of message from the message-queue.

**Augments:** Message

**Unsubscribe:** Lets AXE know that the plug-in is no longer interested in this kind of messages from the message-queue.

**Author:** Gives the authors of the plug-in.

**Returns:** string[]

**Context:** Gives what context the plug-in is usable.

**Returns:** ContextType[]

**Description:** Gives a discription of the plug-in.

**Returns:** string

**Name:** Gives the name of the plug-in.

**Returns:** string

**PluginDirectoryPath:** Gives the path to the plug-in.

**Return:** String

**Version:** Gives the version of the plug-in.

**Returns:** string

**GetConfigurationUserControl:** Gives the GUI-form for the configuration of plug-in.

**Returns:** UserControl

**SavePluginConfiguration:** Lets the plug-in know to save the configuration.

**Augments:** :UserControl

**Load:** The function that initializes the plug-in.

**Unload:** Disposes of the plug-in.

## The Controller's Auxiliary Methods

The auxiliary methods in the *Controller* class.

***MainMessageParsingLoop***: The method that is called from platform whenever a message is broadcasted on the message-queue, if the implementing controller subscribe to the broadcasted message type. It parses the message and calls the method needed.

***AllocateThread***: Allocates a thread from AXE's thread-pool.

**Augments**: DocumentHandler

***MessageParsingExternal***: A method that parses any messages from the message-queue that is not caught by the *MainMessageParsingLoop*.

**Augments**: MessageType

***PlatformFileOpenExtendPoint***: A method used in the implementing controllers to add code to be executed when a file is opened.

**Augments**: MessageType

***PlatformFileCloseExtendPoint***: A method used in the implementing controllers to add code to be executed when a file is closed.

**Augments**: MessageType

***PlatformViewTabChangedExtendPoint***: A method in the implementing controller to add code to be executed when the when view-tab on AXE is changed.

**Augments**: MessageType

***SpawnDocumentHandler***: A method that returns a specialized DocumentHandler.

**Augments**: IPlugin, Stream

**Returns**: Documenthandler

## 4.5.2 Common DocumentHandler

The functionality that this class provides is as stated in **Section 3.3 - Common** to manage open files. The common *DocumentHandler* acts as a handle to the open files. An overview of methods and attributes in the class can be seen in **Figure 4.4**.

<i>DocumentHandler</i>
alive Stream Parent
SendMessage() StartThread()

Figure 4.4: Complete class diagram of *DocumentHandler*.

### The DocumentHandler's Methods

A document Handler contains the following methods.

***SendMessage:*** Send a message to the message queue through the use of the common controller.

***DocumentHandlerThreadStart:*** Starts a thread for the associated document handler.

### Common Status

This is an auxiliary class that is use to send messages about the status of a plug-ins progress and return status.

The class diagram for the Status looks as on **Figure 4.5**.

Status
procentDone statusValue

Figure 4.5: Class diagram for Status.

## Common Message

This class is the class broadcasted to the message queue. It contains methods to set various information that is thought needed by other plug-ins to get useful information about another plug-ins information. **Figure 4.6** shows an overview of the class.

Message
content
type
MessageType() Content()

Figure 4.6: Class diagram for Message.

## Common Error

The class contain the errors from a plug-in functional process. The Error class contains methods and attributes to set and store information about where errors have occurred in an XML-document. **Figure 4.7**

Error
content
line
title
offset
to

Figure 4.7: Class diagram for Error.



Figure 4.8: Complete class diagram of Controller

## 4.6 Well-formedness Components

This section documents the design decisions made during the development of the Well-formedness Checker plug-in. How the plug-in should communicate with the platform. And how the well-formedness functionality is provided.

### 4.6.1 Control Component

The Control Component of the Well-formedness Checker consist of the Well-formedness Controller and Well-formedness DocumentHandler classes. The class-diagram, see **Figure 4.9**, shows the connection between the Well-formedness Checker control components and the common components described in **Section 4.5 - Common Component** . The class diagram for the Well-formedness Checker components uses a notation that differs a little from standard UML notation. The difference is that in methods that are overridden in an implementing class are grayed out in the super class.

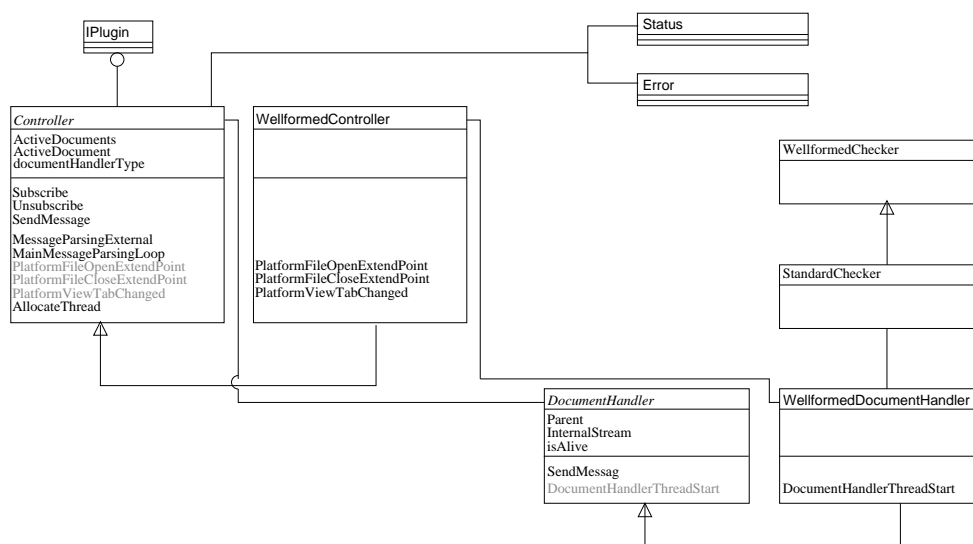


Figure 4.9: The Well-formedness Checker classes.

### Description of the Components Classes

**Well-formedness Controller** The class `WellformedController` is the main class of the Well-formedness Checker plug-in. The class extends the class `Controller` and handles the subscription and unsubscription of messages from the platform. The class also handles the creation and updating of the Well-formedness Checkers user interface and its document handlers. A class diagram for the class can be found in **Figure 4.10**.

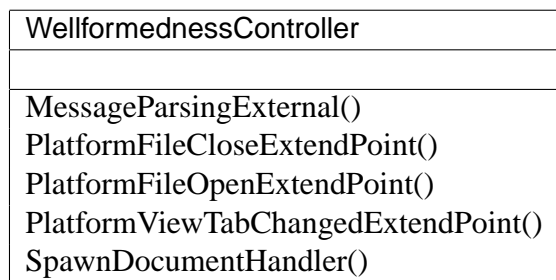


Figure 4.10: Class diagram for WellformednessController.

**Well-formedness DocumentHandler** The WellformednessDocumentHandler class extends *DocumentHandler*. The responsibility of the class is to initiate well-formedness check (the actual well-formedness checking) on each individual file associated to the plug-in (all open XML files). A class diagram for the class can be found in **Figure 4.11**.

Each instance of WellformedDocumentHandler is run within its own thread in order to allow files to be processed in parallel.

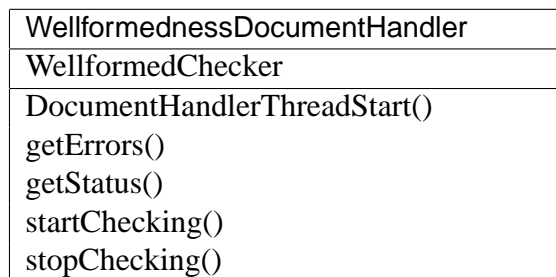


Figure 4.11: Implementation class for WellformednessDocumentHandler.

## 4.6.2 User Interface Component

The user interface consists of one class the WellformedGUI

### Description of the Components Classes

**WellformedGUI** The class WellformedGUI extends the .NET framework class *UserControl* and is responsible for showing the well-formedness status of the active document. **Figure 4.12** shows an overview of the class its methods and attributes.

The status of the document falls into one of the following states:

- Document is well-formed.
- Document status is unknown (processing).

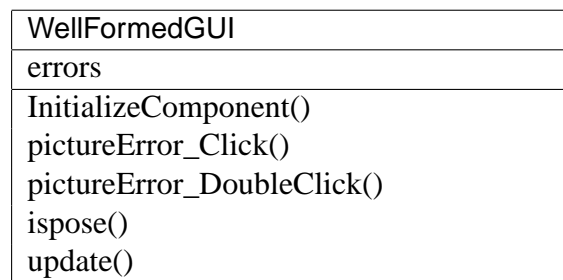


Figure 4.12: Class diagram for WellFormedGUI.

- The document is not well-formed.

The above mentioned states are visually represented by an icon on the status bar of the platform. The icon represents the status of the active document by the following convention:

- Green - Document is well-formed.
- Grey - Document status is unknown (processing).
- Red - The document is not wellformed.

In order to make the icon viewable for persons with other than normal color perception the icons are made to be different even when viewed in grayscale. an example of the icons can be seen on **Figure 4.13**.



Figure 4.13: The three status icons.

### 4.6.3 Functional Component

The functional component of the Well-formedness Checker has two different ways to check the well-formedness of an XML-document. The two differs in complexity. The simplest, called the standard checker, of the two approaches will be described in the coming section. Discussion of the complex algorithm is postponed until **Chapter 6 - Continuous Checking of Large Files** . To ensure conformance between different implementations of Well-formedness Checkers all shared functionality is put into the abstract class *WellformednessChecker*.

### Description of the Components Classes

**Well-formedness Checker** This class is the abstract class used to ensure conformance between different implementation of Well-formedness Checker algorithms. The well-formedness checkers will need a stream representing the XML-document. **Figure 4.14**

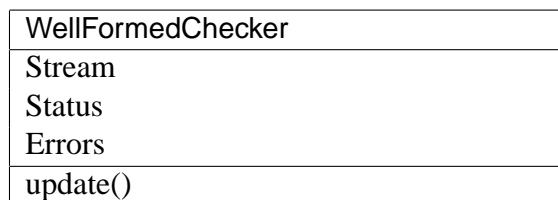


Figure 4.14: Class diagram for WellformedChecker.

**The Standard Checker** The simple Well-formedness Checker utilizes the well-formedness checker implemented by Microsoft in the XML package for .NET. **Figure 4.15** show the methods and attributes of the class.

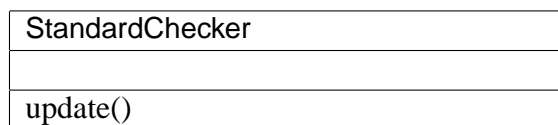


Figure 4.15: Class diagram for StandardChecker.

**The Complex Checker** The complex Well-formedness Checker uses the algorithm described in **Chapter 6 - Continuous Checking of Large Files** . The Algorithm works by keeping track of which parts an XML-document that are already well-formed. If a part change the algorithm only check the well-formedness of that part.

## 4.7 Validation Components

This section documents the design decisions made during the design of the Validator plug-in. Decisions concerning how to communicate with the platform. And how the validation functionality is provided.

### 4.7.1 Control Components

The control component of the Validator consist of the ValidationController and the ValidationDocumentHandler classes. The class diagram, see **Figure 4.16**, shows that the validations control components extends the common control components. Again the notation in the class diagram differs from the standard UML notation.

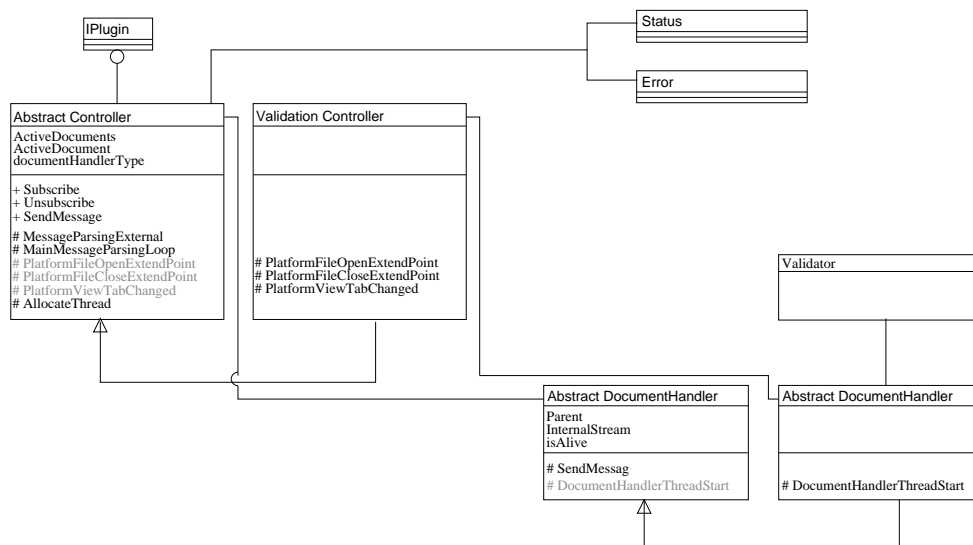


Figure 4.16: Class diagram for the Validator.

### Description of the Components classes

**Validation Controller** This is the main class of the Validation plug-in. The class extends the abstract class controller and its responsibility is to handle subscription and unsubscription of platform messages and to instantiate the appropriate number of ValidationDocumentHandler objects. The class is also responsible of creating the user-interface which consists of a menu for starting the validation process and a side pane for showing the validation results. **Figure 4.17**

**Validation DocumentHandler** This class extends the abstract class DocumentHandler and each instance is responsible for handling a single document and initiating the val-

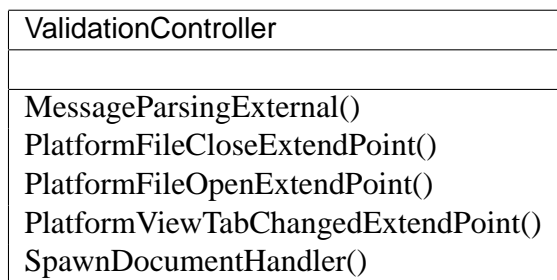


Figure 4.17: Class diagram for ValidationController.

validation of such. Each validationDocumentHandler is run in a new thread in order to allow parallel processing of multiple files. **Figure 4.18** show the class with methods.

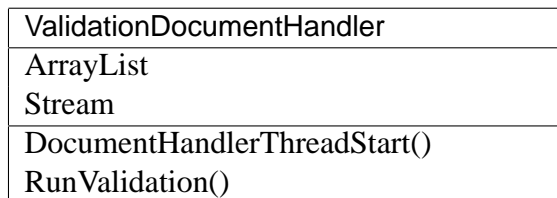


Figure 4.18: Class diagram for ValidationDocumentHandler.

### 4.7.2 User Interface Component

The user interface of the validator consists of the three classes each with a very well-defined responsibility. How the user interface is graphically layered see **Figure 4.19**.

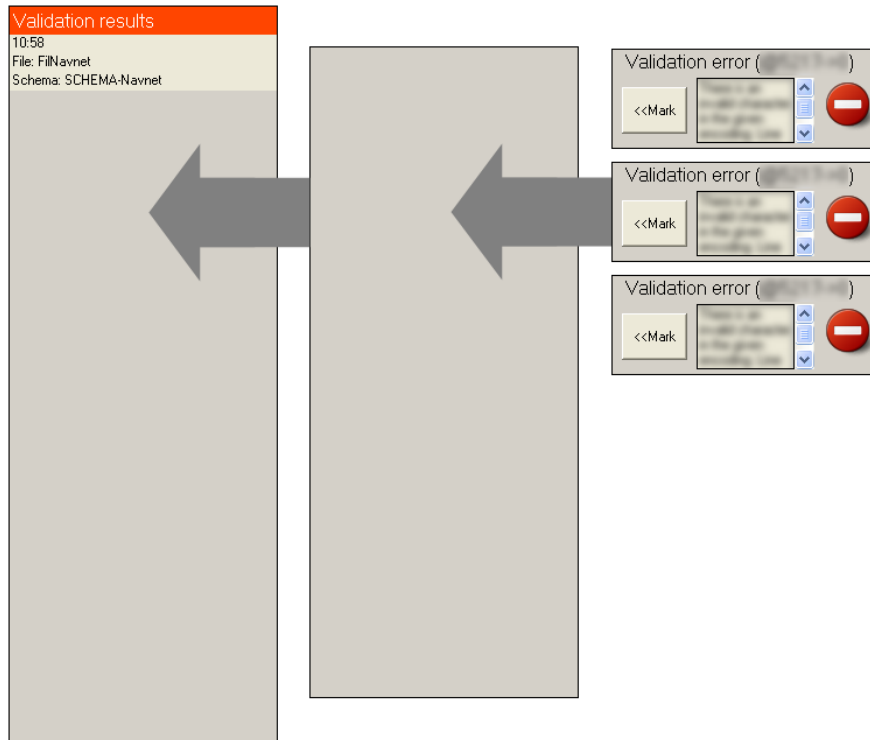


Figure 4.19: The Validation GUI.

#### Description of the Components Classes

**ValidationGUI** The class ValidationGUI acts as a placeholder in witch all information is presented to the user. This class extends the class UserControl. **Figure 4.20**

ValidationGUI
Container
Label
Panel
ValidationErrorGUI
InitializeComponent() validationErrorGUI1_Load()

Figure 4.20: Class diagram for ValidationGUI.

**ValidationErrorGUI** The class ValidationErrorGUI is a class used to hold a set of ValidationError and is responsible for creating the appropriate scroll-bars if there are more errors in the document than can be viewed in the side pane. **Figure 4.21**

ValidationErrorGUI
Container ValidationError[]
update()

Figure 4.21: Class diagram for ValidationErrorGUI.

**ValidationError** This is the class actually displaying the error information to the user. The class is also responsible for providing the user with the option to mark errors in other views by requesting this via the message queue. **Figure 4.22**

ValidationError
Button components Label Error
InitializeComponent() button1_Click()

Figure 4.22: Class diagram for ValidationError.

### 4.7.3 Functional Component

This subsection describe the working class of the Validator plug-in. It gives a short description of which methods and attributes the class has. **Figure 4.23** shows the class diagram of the class.

#### Description of the Components

**Validate** This is the class that does the actual validation. The validator used is the XML validating reader from System.XML a package in the Microsoft .NET framework.

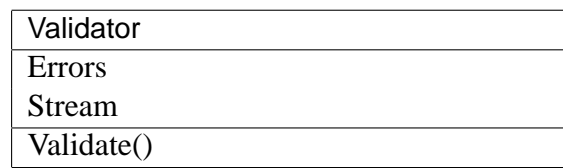


Figure 4.23: The Validate Class.

## 4.8 XPath Components

This section documents the design decisions made during the well-formedness plug-in. The Communication with the platform and the functionality of the plug-in is discussed.

### 4.8.1 Control Component

The control component of the XPath plug-in consists of the two classes, XPathController and XPathDocumentHandler. The class diagram, see **Figure 4.24**. The control component of the plug-in is the part of the plug-in that interacts with the platform. Furthermore the controller controls the program flow and updating of the user interface.

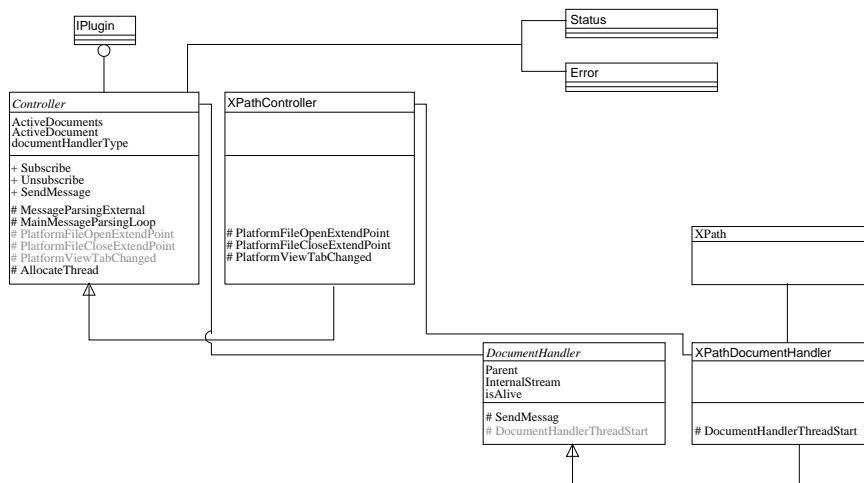


Figure 4.24: Class diagram for XPathController.

### Description of the Components Classes

**XPath Controller** The class `XPathController` is the main class of the plug-in. This class extends the abstract class `Controller` and processes the messages to and from the platform and the user.

The `XPathController` also controls updating of the GUI and spawns an instance of the

`XPathDocumentHandler` in its own thread each time the platform opens an XML document.

**XPath DocumentHandler** The `XPathDocumentHandler` extends the abstract class `DocumentHandler` and it serves the role of managing operations related to one specific document (eg. starting the actual XPath search on the document)

## 4.8.2 User Interface Component

### Description of the Components Classes

**XPathGUI** The class `XPathGUI` extends the .NET class `UserControl` and acts as a placeholder for the components of the GUI. Objects of the class is added to the side pane of the platform in order to allow user interaction.

**XPathQueryGUI** This class is a GUI component used to allow the user to write an XPath query and run this on the active document.

**XPathResultGUI** The `XPathResultGUI` is a GUI component extending the `UserControl` class. Each instance of the class should be used to represent a single result from a query. The component contains a small section of text surrounding the query match, and when clicked the section should be marked in the editor via the message queue.

If the query returns several results, several instances of the class should be added to the side pane allowing the user to mark each result in the editor view.

## 4.8.3 Functional Component

### Description of the Components Classes

**XPath** The functional component consists of a single class `XPath` which is responsible for executing the actual query on the active document. The class uses the .NET framework's implementation of XPath, since a full implementation of XPath is outside the scope of our project.

## 4.9 DTD2Schema

The DTD2Schema plug-in was rejected in the design phase since the original motivation behind the existence of the plug-in no longer existed. The motivation behind the ability of converting DTD to Schemas was that the validation plug-in only had to handle Schemas, and therefore would be simpler to implement.

However when the XML capabilities in the .NET framework was discovered (it could handle DTD and Schemas) the development of the DTD2Schema converter was terminated.

## 4.10 Implementation Plan

At the end of the design phase an implementation plan was discussed within the group.

The discussion did not result in a formal plan but in a general consensus that a running version of the validation and well-formed plug-ins had higher priority than refinements on these plug-ins. The other plug-ins were given a lower priority.

The priorities were motivated by the priorities given in **Appendix A - System Requirements Specification** and because the group anticipated that developing the plug-ins under the conditions set by the development structure, the technical environment, and the including four separate groups had the potential to slow down the implementation considerably.



# Chapter 5

## Implementation

The AXE platform plug-ins is implemented in C# using Microsoft Visual Studio .NET (as required in the specification). The plug-ins, Well-formedness Checker and Validator is implemented by extending and using the classes in the Common component. Extending and using the classes from the common component minimize the number of times the communication with the platform has to be implemented.

### 5.1 Common Components Implementaion Issues

The Common component consists of the following classes *Controller*, *DocumentHandler*, *Error*, *Status* and *Message*.

**Using Common** As laid out in **Chapter 3 - Analysis** and **Chapter 4 - Design** the plug-ins' controller implements the common *Controller*. Each of the abstract methods provided in the abstract *Controller* must be overloaded. Overloading of abstract is done in C# by using the keyword `override`. To get the *Controller* working on both the abstract methods specified in the common *Controller* and the *IPlugin* interface both have to be overridden. The two following listing show the abstract methods of the *Controller* and the *IPlugin* interface that must be overridden:

The *MessageParsingExternal* method provide the implementing controller with the ability to handle messages on the message queue not handled in the abstract *Controller*. The abstract *Controller* then parses the message unknown onto the overridden *MessageParsingExternal* in the *Plug-inController* to which the message could be known.

#### 5.1.1 Controller

The implementation structure for *Controller* class can be seen on **Listing 5.1**.

```
1 | // Class .
```

```

public abstract Controller
3 // Constructors .
public Controller ()
5 // Fields .
private System.Type documentHandlerType
public Axe.Plugin.Common.DocumentHandler
    ActiveDocument
public System.Collections.Hashtable
    ActiveDocuments
// Properties .
10 public string[] Author
public Axe.DataClasses.ContextType[] Context
public string Description
public System.Type DocumentHandlerType
public string Name
15 public string PluginDirectoryPath
public string Version
// Methods .
protected void Unsubscribe(Axe.Plugin.Common.
    Message message)
protected void MainMessageParsingLoop(Axe.
    Managers.MessageQueue.MessageType type , Axe.
    Managers.MessageQueue.MessageContent content)
20 protected void MessageParsingExternal(Axe.
    Managers.MessageQueue.MessageType type , Axe.
    Managers.MessageQueue.MessageContent content)
protected void PlatformFileCloseEndPoint(Axe.
    Managers.MessageQueue.MessageType type , Axe.
    Managers.MessageQueue.MessageContent content)
protected void PlatformFileOpenEndPoint(Axe.
    Managers.MessageQueue.MessageType type , Axe.
    Managers.MessageQueue.MessageContent content)
protected void PlatformViewTabChangedEndPoint(
    Axe.Managers.MessageQueue.MessageType type ,
    Axe.Managers.MessageQueue.MessageContent
    content)
protected void AllocateThread(Axe.Plugin.Common.
    DocumentHandler document)
25 protected Axe.Plugin.Common.DocumentHandler
    SpawnDocumentHandler(Axe.Interfaces.IPlugin
    iplugin , System.IO.Stream stream)
protected void Subscribe(Axe.Plugin.Common.
    Message message)
public void AddAToolBarButton(System.Drawing.
    Image icon , System.EventHandler del , Axe.
    DataClasses.ContextType[] contextType)
public void AddIconPanel(System.Drawing.Icon icon
    , Axe.DataClasses.ContextType[] contextType ,
    string tooltipText)
public Axe.Managers.StatusBarManager.
    AProgressBarTask AddProgressBar(string
    taskName , int numberOfSteps)
30 public void AddSidePane(Axe.Managers.
    SidePaneManager.Pane pane)

```

```

32 // Methods .
   public void AddSubMenuItem(Axe.Managers .
       MenuManager.AMenuItem aMenuItem , string
       menuText , Axe.DataClasses.ContextType []
       contextType , System.EventHandler del , Axe.
       Managers.MenuManager.ShortCut emacsSC , Axe.
       Managers.MenuManager.ShortCut windowsSC)
   public void AddViewTab(Axe.Interfaces.View view ,
       string viewName)
   public System.Windows.Forms.UserControl
       GetConfigurationUserControl ()
35 public void Load ()
   public void PostMessageOnStatusBar (string message
       )
   public void SavePluginConfiguration (System .
       Windows.Forms.UserControl userCtrl)
   public void SendMessage (Axe.Plugin.Common.Message
       message)
   public void SetProjectManager ()
40 public Axe.Managers.MenuManager.AMenuItem
       MenuItem (string menuText , Axe.DataClasses .
       ContextType [] contextType , System.EventHandler
       del , Axe.Managers.MenuManager.ShortCut
       emacsSC , Axe.Managers.MenuManager.ShortCut
       windowsSC)
   public void Unload ()
   public void UpdateGUI (Axe.Plugin.Common.Status
       status , Axe.Plugin.Common.Error [] errors)
   public void UpdateProgressBar (Axe.Managers .
       StatusBarManager.AProgressBarTask task)

```

Listing 5.1: Implementation for *Controller* class.

Implementing the basics of the *Controller* was without any problems. The more advanced aspects such as maintaining the correct *ActiveDocument* and passing on messages to the concrete Controllers and DocumentHandlers proved to be harder.

### Known Issues

**Passing Messages Down** Sometimes messages do not get passed down to eg. *FileOpenEx-tentPoint* in the concrete classes. Since this does not always happen it could be a concurrency issues. Since the bug isn't readily reproducible we have not been able to locate the problem in the source.

## 5.1.2 DocumentHandler

The implementation structure for *DocumentHandler* class can be seen on **Listing 5.2**.

```

1 // Class .
   public abstract class DocumentHandler

```

```

// Constructors .
public DocumentHandler()
5 public DocumentHandler(Axe.Plugin.Common.
    Controller Parent , System.IO.Stream
    InternalStream )
public DocumentHandler(System.IO.Stream stream)
// Fields .
private bool isAlive
protected System.IO.Stream internalStream
10 protected Axe.Plugin.Common.Controller Parent
// Properties .
public bool alive
public System.IO.Stream InternalStream
// Methods .
15 protected void SendMessage(Axe.Plugin.Common.
    Message Message)
public void DocumentHandlerThreadStart()

```

Listing 5.2: Implementation for *DocumentHandler* class.

The problems encountered during the implementation of the *Controller* and *DocumentHandler* classes are hard to separate. The implementation problems are discussed for the classes. Parallel to the discussion about implementation problems the use of common is discussed.

**Communication with Platform** Communicating with the platform was difficult in some cases. Interfaces and tutorials for this was publicized early in the project but the interfaces changed frequently and some did not work as expected and the tutorials were never updated. The frequent changes in the interfaces to the platform resulted in the decision to lock a build. The Well-formedness Checker and Validator plug-ins were targeted at the chosen build. The nightly build of the third of December were chosen.

Locking the build made it possible to focus on generating new source instead of adapting the existing code to changing interfaces.

**Spawning new DocumentHandlers** The *Controller* is responsible for creating new *DocumentHandlers* when new files are opened. This resulted in problems because the abstract *Controller* does not know what kind of specialized *DocumentHandler* it should create when requested to. Early versions used reflection to determine which kind document-handler to be created. However this turned out to be time consuming and unviable. Instead the specialized *Controller* has to do a call back with the specialized *DocumentHandler*.

```

1 protected void MainMessageParsingLoop(Axe.
    Managers.MessageQueue.MessageType type , Axe.
    Managers.MessageQueue.MessageContent content)
{

```

```

5  switch ( type.Name)
6  {
    case " Platform.File.Open":
        // decode the content to get which file has
        // been opened, and add it to the
        // HashTable
        if ( content.Content.GetType() == typeof(Axe
            .Managers.ViewTabManager.ViewTab) )
10     {
        Axe.Managers.ViewTabManager.ViewTab tab
            = (Axe.Managers.ViewTabManager.
                ViewTab) content.Content;
        // Get the stream from content
        System.IO.Stream stream = tab.View.Data.
            Stream;
        // now make a new document handler
        DocumentHandler documentHandler = this.
            SpawnDocumentHandler(this, stream);
15     // Now make a new thread for the document
        // handler
        this.AllocateThread(documentHandler);
        // next we need to put the
        // documenthandler into the Hashtable
        int hashKey = documentHandler.GetHashCode
            ();
        ActiveDocuments.Add(hashKey,
            documentHandler);
20     // Set this file to the active document
        this.ActiveDocument = documentHandler;
        this.PlatformFileOpenEndPoint(type,
            content);
    }
    break;
25 [snip]

```

Listing 5.3: Controller.cs:MainMessageParsingLoop

```

1  protected override SpawnDocumentHandler
    SpawnDocumentHandler(IPlugin iplugin, System.
        IO.Stream stream)
    {
        return new WellformednessDocumentHandler(this,
            stream);
    }

```

Listing 5.4: WellformednessController.cs:SpawnDocuementHandler

The instantiation the *DocumentHandler* object starts in the *MainMessageParsingLoop* (see **Listing 5.3** line 14 -> end) then calls the *SpawnDocumentHandler* in the WellformednessController (used as example). Since the WellformednessController (see **Listing 5.4**) knows which type of *DocumentHandler* is needed to be created and reflections are avoided.

### 5.1.3 Error

The implementation structure for Error class can be seen on **Listing 5.5**.

```

1 // Class .
  public class Error
  // Constructors .
  public Error(string title , string content , ulong
    line , ulong offset)
5 // Fields .
  public string content
  public ulong line
  public ulong offset
  public string title
10 // Properties .
  // Methods .

```

Listing 5.5: Implementation for Error class.

The implementation of the Error class is as can be seen rather simple, it consists of set of fields and a constructor.

### 5.1.4 Status

The Status is just a small container clas and there where no issues implementing this. The implementation structure for Status class can be seen on **Listing 5.6**.

```

1 // Class .
  public class Status
  // Constructors
  public Status ()
5 // Fields .
  public int procentDone
  public Axe.Plugin.Common.StatusValue statusValue
  // Properties .
  // Methods .

```

Listing 5.6: Implementation for Status class.

### 5.1.5 Message

The Message is just a simple class used to contain data. The implementation structure for Message class can be seen on **Listing 5.7**.

```

1 // Class .
  public class Message
  // Constructors .
  public Message(string messageType , object content
    )
5 // Fields .
  private object content

```

```
8  private string type  
   // Properties .  
10 public object Content  
   public string MessageType  
   // Methods .
```

Listing 5.7: Implementation for Message class.

## 5.2 Well-formedness Plug-in Implementation Issues

The responsibility of the well-formed checker is to check the active document for well-formedness when required

This is implemented by catching the following messages from the message queue.

- Platform.File.Open
- Plugin.XML.Data.Changed

When one of these messages is caught the plug-in starts to check the document for well-formedness.

### 5.2.1 WellformednessController

Well-formedness Controller and DocumentHandler uses the methods inherited from the abstract classes in order to function as a plug-in

The implementation structure for WellformednessController class can be seen on **Listing 5.8**.

```

1 // Class .
  public class WellformednessController
  // Constructors
  public WellFormednessController ()
5 // Fields .
  private Axe.Managers.StatusBarManager .
    AStatusBarPanel asbp
  private string [] authors
  private Axe.DataClasses.ContextType [] context
  private string description
10 private string name
  private string pluginDirectoryPath
  private string version
  // Properties .
  public string [] Author
15 public Axe.DataClasses.ContextType [] Context
  public string Description
  public string Name
  public string PluginDirectoryPath
  public string Version
20 // Methods .
  protected void MessageParsingExternal (Axe.
    Managers.MessageQueue.MessageType type , Axe.
    Managers.MessageQueue.MessageContent content )
  protected void PlatformFileCloseExtendPoint (Axe.
    Managers.MessageQueue.MessageType type , Axe.
    Managers.MessageQueue.MessageContent content )
  protected void PlatformFileOpenExtendPoint (Axe.
    Managers.MessageQueue.MessageType type , Axe.
    Managers.MessageQueue.MessageContent content )

```

```

protected void PlatformViewTabChangedEndPoint(
    Axe.Managers.MessageQueue.MessageType type ,
    Axe.Managers.MessageQueue.MessageContent
    content )
25 protected Axe.Plugin.Common.DocumentHandler
    SpawnDocumentHandler(Axe.Interfaces.IPlugin
    iplugin , System.IO.Stream stream)
public void SavePluginConfiguration(System.
    Windows.Forms.UserControl userCtrl)
public void Unload()
public void UpdateGUI(Axe.Plugin.Common.Status
    status , Axe.Plugin.Common.Error[] errors)
public void CreateNewThread(Axe.Managers.
    ThreadManager.ThreadPool.Delegate del)
30 public System.Windows.Forms.UserControl
    GetConfigurationUserControl()
public void Load()

```

Listing 5.8: Implementation for WellformednessController class.

### 5.2.2 WellformednessDocumentHandler

The WellformednessDocumentHandler follows the design from **Section 4.6.1**.

The implementation structure for WellformedDocumentHandler class can be seen on **Listing 5.9**.

```

1 // Class .
public class WellformednessDocumentHandler
// Constructors
public WellformednessDocumentHandler(Axe.Plugin.
    Common.Controller Parent , System.IO.Stream
    inStream)
5 public WellformednessDocumentHandler()
// Fields .
private Axe.Plugin.WellFormedness.
    WellformedChecker checker
// Properties .
// Methods .
10 public void DocumentHandlerThreadStart()
public Axe.Plugin.Common.Error[] getErrors()
public Axe.Plugin.Common.Status[] getStatus()
public void startChecking()
public void stopChecking()

```

Listing 5.9: Implementation for WellformednessDocumentHandler class.

### 5.2.3 WellformedChecker

The WellformedChecker follows the design from **Section 4.6 - Well-formedness Components**

The implementation structure for WellformedChecker class can be seen on **Listing 5.10**.

```

1 // Class .
  public class WellformedChecker
  // Constructors
  public WellformedChecker(System.IO.Stream
    inputStream)
5 // Fields .
  protected System.IO.Stream inputStream
  public Axe.Plugin.Common.Status status
  public Axe.Plugin.Common.Error[] errors
  // Properties .
10 // Methods .
  public void run()
  public void update(Axe.Plugin.Common.Message msg)
  public void update(System.IO.Stream inputStream)
  public void UpdateEndPoint()

```

Listing 5.10: Implementation for WellformedChecker class.

## 5.2.4 StandardChecker

The implementation structure for StandardChecker class can be seen on **Listing 5.11**.

```

1 // Class .
  public class StandardChecker : WellformedChecker
  // Constructors
  private StandardChecker()
5 public StandardChecker(System.IO.Stream
    inputStream)
  // Fields .
  private int COUNT
  // Properties .
  // Methods .
10 public void run()
  public void update(Axe.Plugin.Common.Message msg)
  public void UpdateEndPoint()

```

Listing 5.11: Implementation for StandardChecker class.

## 5.2.5 WellFormedGUI

The main user-interface for the well-formed checker is a small icon on the status bar which displays the current status of the active document.

The implementation structure for class WellFormedGUI can be seen on **Listing 5.12**.

```

1 // Class .
  public class WellFormedGUI
  // Constructors
  public WellFormedGUI()

```

```
5 public WellFormedGUI(Axe.Plugin.Common.Status
    status , Axe.Plugin.Common.Error[] errors)
6 public WellFormedGUI(Axe.Plugin.Common.Status
    status)
    // Fields.
    private System.ComponentModel.Container
        components
    private Axe.Plugin.Common.Error[] errors
10 private System.Windows.Forms.PictureBox
    pictureError
    private System.Windows.Forms.PictureBox pictureOk
    private System.Windows.Forms.PictureBox
    pictureUnknown
    private System.Windows.Forms.ToolTip toolTipError
    private System.Windows.Forms.ToolTip toolTipOk
15 private System.Windows.Forms.ToolTip
    toolTipUnknown
    // Properties.
    // Methods.
    private void InitializeComponent()
    private void pictureError_Click(object sender ,
        System.EventArgs e)
20 private void pictureError_DoubleClick(object
    sender , System.EventArgs e)
    protected void Dispose(bool disposing)
    public void update(Axe.Plugin.Common.Status
        status)
```

Listing 5.12: Implementation for WellFormedGUI class.

This status icon is placed on the status-bar of the platform and a tool tip is assigned in order to explain the status further. Furthermore if the user double-clicks on the icon when it is red, a pop-up with a more detailed description of the error is displayed. At the same time a message is sent requesting the error to be marked in the XML Editor plug-in.

### Known Issues

**Updating Status Icon** The Well-formedness Checker use a pop-up when it is updating the status icon otherwise the platform will crash. It seems as though the platform cannot handle rapid GUI updates. (This problem has been solved by the platform group in the latest platform version, but it is not compatible with our plug-in due to changes in the platform interface.)

**Closing all Files** When closing the last XML file in the platform, the status icon does not vanish as it should.

## 5.3 Validation Component Implementation Issues

### 5.3.1 ValidationController

The implementation structure for ValidationController class can be seen on **Listing 5.13**.

```

1  // Class .
   public class ValidationController : Controller
   // Constructors .
   public ValidationController ()
5  // Fields .
   private string [] authors
   private Axe.DataClasses.ContextType [] context
   private string description
   private string name
10  private string pluginDirectoryPath
   private string version
   // Properties .
   public string [] Author
   public Axe.DataClasses.ContextType [] Context
15  public string Description
   public string Name
   public string PluginDirectoryPath
   public string Version
   // Methods .
20  private void ValidationEvent(object Sender ,
      System.EventArgs click )
   protected void MessageParsingExternal(Axe.
      Managers.MessageQueue.MessageType type , Axe.
      Managers.MessageQueue.MessageContent content )
   protected void PlatformFileCloseEndPoint(Axe.
      Managers.MessageQueue.MessageType type , Axe.
      Managers.MessageQueue.MessageContent content )
   protected void PlatformFileOpenEndPoint(Axe.
      Managers.MessageQueue.MessageType type , Axe.
      Managers.MessageQueue.MessageContent content )
   protected void PlatformViewTabChangedEndPoint(
      Axe.Managers.MessageQueue.MessageType type ,
      Axe.Managers.MessageQueue.MessageContent
      content )
25  protected Axe.Plugin.Common.DocumentHandler
      SpawnDocumentHandler(Axe.Interfaces.IPlugin
      iplugin )
   public System.Windows.Forms.UserControl
      GetConfigurationUserControl ()
   public void Load ()
   public void SavePluginConfiguration (System.
      Windows.Forms.UserControl userCtrl )
   public void Unload ()
30  public void UpdateGUI(Axe.Plugin.Common.Status
      status , Axe.Plugin.Common.Error [] errors )

```

Listing 5.13: Implementation class for ValidationController.

The only implementation issue for this component were in getting the *Controller* working for the concrete Controller.

### Knows Issues

**Starting Validator** In order to start validation on a menu selection the menu sends a message “Plugin.Validation.Start” on the MessageQueue. This is done in order to avoid additionally inter-thread communication between the menu (in the platform’s thread) and the documentHandler (in its own thread).

Preventing the Validator from starting when an XML-file is opened, it was needed to start it from a menu. This proved to be tricky and was finally done by having the menu sending the message “Plugin.Validation.Start” on the MessageQueue.

## 5.3.2 ValidationDocumentHandler

The implementation structure for ValidationDocumentHandler class can be seen on **Listing 5.14**.

```
1 // Class .
  public class ValidationDocumentHandler :
      DocumentHandler
  // Constructors .
  public ValidationDocumentHandler ()
5  public ValidationDocumentHandler (Axe.Plugin .
      Common.Controller Parent , System.IO.Stream
      inStream )
  // Fields .
  public System.Collections.ArrayList al
  private System.IO.Stream stream
  // Properties .
10 // Methods .
  public void DocumentHandlerThreadStart ()
  public void RunValidation ()
```

Listing 5.14: Implementation for ValidationDocumentHandler class.

## 5.3.3 Validator

This class performs the actual validation using a Microsoft .NET framework library.

The implementation structure for Validator class can be seen on **Listing 5.15**.

```
1 // Class .
  public class Validator
  // Constructors .
  private Validator ()
5  public Validator (System.IO.Stream stream)
  // Fields .
```

```

private System.Collections.ArrayList errors
private System.IO.Stream inputStream
9 // Properties.
10 // Methods.
private void ValidationCallBack(object sender ,
    System.Xml.Schema.ValidationEventArgs args)
public System.Collections.ArrayList Validate()

```

Listing 5.15: Implementation for Validator class.

## Known Issues

**Validating non-Wellformed Files** If you validate an XML-file that is not wellformed the Validator will return that the file is valid.

### 5.3.4 ValidationGUI

The main GUI class for displaying the validation result. The constructor of this class creates all the elements of the GUI, either by it self or by calling the other GUI classes.

The implementation structure for ValidationGUI class can be seen on **Listing 5.16**

```

1 // Class.
public class ValidationGUI
// Constructors.
public ValidationGUI(string fileName , string
    schemaName , ArrayList errors)
5 // Fields.
private System.ComponentModel.Container
    components
private System.Windows.Forms.Label label1
private System.Windows.Forms.Label labelFileName
private System.Windows.Forms.Label
    labelSchemaName
10 private System.Windows.Forms.Label labelTime
private System.Windows.Forms.Panel panell
private Axe.Plugin.Validation.ValidationErrorMessageGUI
    validationErrorMessageGUI
// Properties.
// Methods.
15 protected void Dispose(bool disposing)
private void InitializeComponent()
private void validationErrorMessageGUI_Load(object
    sender , System.EventArgs e)

```

Listing 5.16: Implementation for ValidationGUI class.

### 5.3.5 ValidationErrorMessageGUI

The GUI displayed when the result contains errors.

The implementation structure for `ValidationErrorGUI` class can be seen on **Listing 5.17**.

```

1 // Class .
  public class ValidationErrorGUI
  // Constructors .
  public ValidationErrorGUI()
5 // Fields .
  private System.ComponentModel.Container
    components
  private Axe.Plugin.Validation.ValidationError[]
    valErrors
  // Properties .
  // Methods
10 private void InitializeComponent()
  protected void Dispose(bool disposing)
  public void update(System.Collections.ArrayList
    errors)

```

Listing 5.17: Implementation for `ValidationErrorGUI` class.

### 5.3.6 ValidationError

The graphical representation of one individual validation error. This element provides the user with a means to mark the error in other plug-ins via a message on the message queue.

The implementation structure for `ValidationError` class can be seen on **Listing 5.18**.

```

1 // Class .
  public class ValidationError
  // Constructors .
  public ValidationError()
5  public ValidationError(Axe.Plugin.Common.Error
    err)
  // Fields .
  private System.Windows.Forms.Button button1
  private System.ComponentModel.IContainer
    components
  private System.Windows.Forms.Label label1
10 private System.Windows.Forms.PictureBox
    pictureError
  private System.Windows.Forms.PictureBox
    pictureWarning
  private System.Windows.Forms.TextBox textBox1
  private System.Windows.Forms.ToolTip toolTip1
  public Axe.Plugin.Common.Error error
15 // Properties .

  // Methods .
  private void button1_Click(object sender, System.
    EventArgs e)
  private void InitializeComponent()
20 protected void Dispose(bool disposing)

```

---

Listing 5.18: Implementation for ValidationError class.

### 5.3.7 ValidationTest

This class is made as a temporary substitute for the not functioning side pane of the platform.

Due to the fact that the side-pane feature of the platform was not working properly at the third of December when the group decided to lock the platform. Resulted in the validation process being displayed in a window(ValidationTest) by itself instead.

The implementation structure for ValidationTest class can be seen on **Listing 5.19**.

```
1 // Class .
  public class ValidationTest
  // Constructors .
  public ValidationTest(string filename , string
    SchemaName , System.Collections.ArrayList
    errors)
5 // Fields .
  private System.ComponentModel.Container
    components
  private Axe.Plugin.Validation.ValidationGUI
    validationGUI1
  // Properties .
  // Methods .
10 private void InitializeComponent()
  protected void Dispose(bool disposing)
```

Listing 5.19: Implementation class for ValidationTest.

## 5.4 XPath Component Implementation Issues

The implementation of the XPath plug-in was chosen not to be moved forward at the start of the implementation phase. The decision was made in order to release resources tied up in the development of the XPath plug-in to the development of the two most important plug-ins in the Plug-in Collection.

The reasoning behind moving the resources was that the validation and well-formed checking functionality was prioritized as “very important” and XPath querying was not even a requirement from the customer see **Appendix A - System Requirements Specification**



# Chapter 6

## Continuous Checking of Large Files

### 6.1 Introduction

#### 6.1.1 The Challenge

A nice feature of AXE would be to be able to give the user instant error-reporting while editing an XML document. That is, whenever the user pauses typing the checking routine should be ready to tell him whether the XML document is correct or if it contains errors. This instant report should allow the user to immediately correct any errors and thereby improve his editing experience.

This feature can be achieved by running a check on the document as soon as the user hesitates to type for a moment. On small documents this moment of hesitation should be enough to run through the entire document and give a reliable report. However, on large documents the checking process will easily take at least a few seconds or maybe even minutes, which is way too much for the user to experience *instant* error-reporting.

And this is exactly the situation, because one of the requirements for this project is that the XML editor should be able to work large XML documents. By “large” is meant documents in the size of 100 MB or more – documents that might not fit main memory.

#### 6.1.2 The Development Task

The task is to develop an optimized approach to checking an XML document, so that the user will continue to experience instantaneous reports when the document grows to huge sizes.

When optimizing the checking process, it is not of main interest how long it takes to check an XML document from beginning to end *once*. The challenge is how to take advantage of the fact that the same document is checked several times.

The setting and the solution idea are like this:

1. Initially, nothing is known about the document, and therefore the document has to be checked from beginning to end. This step is inevitable and can only be done as fast as the fastest checking routine available.
2. But, all subsequent checks should derive benefit from the fact that the document has previously been checked and should, therefore, be able to reuse the results from the preceding check.

An obvious idea of how to utilize the results from the preceding check is to make it possible to check an XML document in parts. Then, in each subsequent check, the checking routine could simply skip any parts that had previously been checked and found correct. This approach will be explained in-depth later on in **Section 6.1.3 - Optimized Checking Strategy** .

Before settling for a solution some experiments were carried out to see what kind of partial checking would be technically feasible.

### **Results from experiments**

Building an XML validator or well-formedness checker from scratch is a huge task by itself, and since the Microsoft .NET Framework already includes classes for these XML tasks it was decided to utilize them as far as possible.

The solution idea was initially tried out with the validation process, because validation provides the most thorough checking of an XML document. However, the validation process is not at all trivial and neither was it to properly do partial validation. A part of an XML document is only valid if it is placed correctly in respect to its context – that is, the rest of the document. Keeping track of this context resolved to be too big a challenge to overcome within reasonable effort.

Stepping down to well-formedness checking it was immediately more feasible to do partial checking. The context of a part of an XML document is not important in respect to well-formedness. To see this, it helps to view XML documents as trees.

### **XML as a tree**

A well-formed XML document is actually a tree-structure and can be viewed as a tree like in **Figure 6.1**. The root element is the root of the tree and the sub-elements of the root are branches. Each sub-element can have its own sub-elements, which are also branches on the tree, and so on. Each branch of the XML tree can also be viewed as a tree itself – an XML sub-tree.

It should be noted that an XML document is by definition well-formed. If it is not well-formed then it is just a textual document that might look like XML. The same goes for XML trees; only well-formed XML is a tree structure. If the XML is not well-formed, the tree brakes.

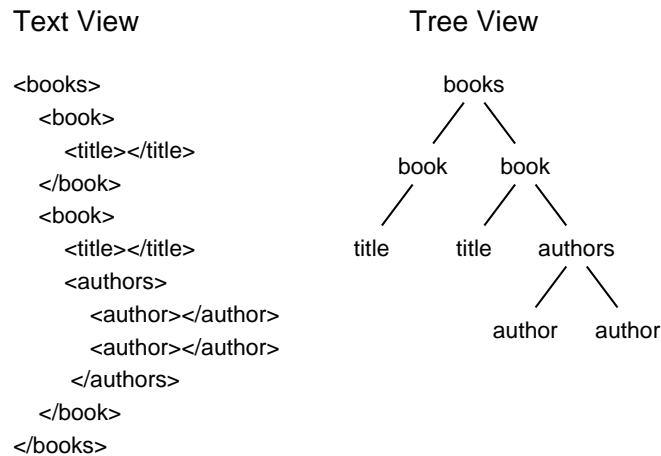


Figure 6.1: Illustration showing how XML can be modeled as a tree.

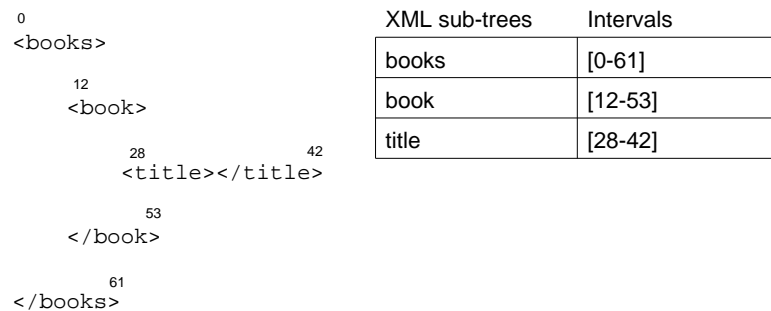


Figure 6.2: Illustration showing how XML sub-trees can be identified as intervals.

Nevertheless, it does make sense to talk about XML documents that are not well-formed. We define these as textual documents that are intended to be XML documents but fails to fulfill the XML well-formedness requirements. Thus, it is assumed that documents edited in an XML editor are intended to be XML documents and will be regarded as such even though they sometimes will not be well-formed.

A well-formed part of a not entirely well-formed XML document is still an XML sub-tree even though the XML document as a whole is not a tree.

### XML sub-trees as intervals

When viewing XML as text, the well-formed sub-trees of a document can be identified as intervals with a begin and end position. For example, the root of a sub-tree is an element with an opening tag and a closing tag. The begin position of the sub-tree interval would then be the position of the first character of the opening tag (the character “<”). Equivalently, the end position of the interval would be the position of

the last character of the closing tag (the character “>”). See illustration in **Figure 6.2**.

Intuitively, the intervals corresponding to the children elements of an XML sub-tree will then be included in the sub-tree interval because they are written between the opening tag and closing tag of the root element of the sub-tree.

### 6.1.3 Optimized Checking Strategy

Returning to the task of achieving partial well-formedness checking, this process can be viewed as checking if all parts of the XML document turn out to be well-formed sub-trees, which together constitute an entirely well-formed XML document.

As mentioned, the well-formed sub-trees can be identified as intervals, and these “well-formed intervals” are the ones we want to store so that they can be skipped the next time the document is checked.

The basic process is like this:

**First check:** Check the document from beginning to end while storing the intervals of all the well-formed sub-trees. If an error is found then report it and stop the checking.

**Subsequent checks:** Check the document from beginning to end but skip the parts corresponding to the stored intervals. Again, store the intervals of any new well-formed sub-trees found. If an error is found then report it and stop the checking. On each check any newly found intervals are added to the intervals from last check.

As described in **Section 3.4.2 - Behavior**, the first check done by the Well-formedness Checker is initiated when a document is opened. All subsequent checks are initiated when the active document is changed. When the document is changed the Well-formedness Checker receives a message about the position of the change. The change is reported as an interval covering the changed part. We call this interval the Change-Interval.

If the Change-Interval overlaps any stored intervals then, basically, these intervals need to be deleted from the stored intervals because it is unknown whether they are still well-formed. However, to optimize performance the Well-formedness Checker checks if there exists a stored interval that includes the Change-Interval. If so then only that interval is checked instead of the whole document. This procedure is detailed later on in **Section 6.2.3 - The Optimized Checker**.

The optimized checking strategy can be illustrated as in **Figure 6.3**. It shows an XML document in a structural view where the graph shows the levels of the elements. The tops are XML sub-trees and the peaks are the leaves. Skipping the the well-formed sub-trees while checking an XML document is like cutting off tops as seen in the illustration depicted as dashed lines.

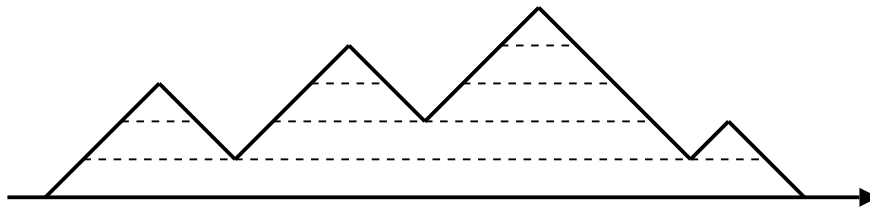


Figure 6.3: Visualization of the optimized checking strategy as cutting off tops in a structural view of an XML document.

## 6.2 Design

The overall analysis and design of how the Well-formedness Checker should integrate and communicate with AXE is already described in **Section 3.4 - Well-formedness** (analysis) and **Section 4.6 - Well-formedness Components** (design). This section deals with the optimized edition of the functional component of the Well-formedness Checker.

The design explains how the following issues should be handled:

- The well-formed intervals has to be stored, retrieved, deleted and updated efficiently.
- The data structure used for storing intervals has to be trimmed so that it does not take up too much memory.
- As the Microsoft .NET XmlTextReader is used for well-formedness checking this tool must be enabled to skip parts of the document it checks.
- The the begin and end position of the well-formed intervals has to be found to be able to store the intervals.

The chapter ends with an overview of how the different parts of the functional component work together to supply the optimized checking strategy.

### 6.2.1 Data Structure for Storing Intervals

During the checking process in the optimized checking strategy the following operations have to be supported by the data structure storing the intervals:

- Store intervals as they are found during the checking process.
- Retrieve intervals during the pass through the XML file as parts have to be skipped.
- Delete intervals when its unclear whether they are still well-formed.

- Update the begin and end position of the intervals when something is deleted from or added to the XML document.

The intervals have a begin and end position and the natural thing for them to represent is the byte position in the XML file where the corresponding well-formed XML subtree begins and ends. However, this means when for example 50 bytes is added to the middle of the XML document then the positions of the intervals coming after this point in the file have to be updated to still point to the correct place in the file. In this case the positions should be increased by 50. This explains the “Update”-operation.

To support the above mentioned operations an Interval Tree data structure has been designed.

### Interval Tree

The Interval Tree data structure is based on the context specific properties of the intervals. As they identify well-formed XML sub-trees, the start and end position can be expected to observe the XML well-formedness rule of proper nesting, thus, they will never partly overlap. This gives the following properties of two intervals  $I1$  and  $I2$  where  $I1.Begin < I1.End$  and  $I2.Begin < I2.End$ :

- $I1.Begin < I2.Begin$  and  $I1.End > I2.End$  –  $I2$  is contained (or fully included) in  $I1$ .
- $I1.Begin < I1.End$  and  $I2.Begin < I2.End$  –  $I2$  comes after  $I1$ , ( $I2$  begins after  $I1$  ends).

The intervals can then be stored in a left-child, right-sibling binary tree as seen in **Figure 6.4**. This implementation uses a shared parent/right-sibling pointer. If a node has a right-sibling then the shared pointer points to that one, else it points to the parent. This has been done to save storage space of the tree and to make it easier to maintain the pointers on insertion and deletion.

Just to clarify the terminology, a “node” in the tree can store an “interval”. A pointer to an interval is really a pointer to the node storing the interval. So, a node and an interval is basically the same, they just emphasize different aspects.

When inserting an interval  $I2$  in relation to  $I1$  the basic insertion rules are:

- If  $I2$  is contained in  $I1$ , then  $I2$  is inserted in the tree as the left-child of  $I1$ .
- If  $I2$  comes after  $I1$ , then  $I2$  is inserted as the right-sibling of  $I1$ .

The following are the maintenance rules to follow when inserting:

- If  $I2$  should be inserted as left-child of  $I1$  and  $I1$  already has a left-child, then  $I2$  is passed on to the left-child of  $I1$ , where it is inserted according to these rules.

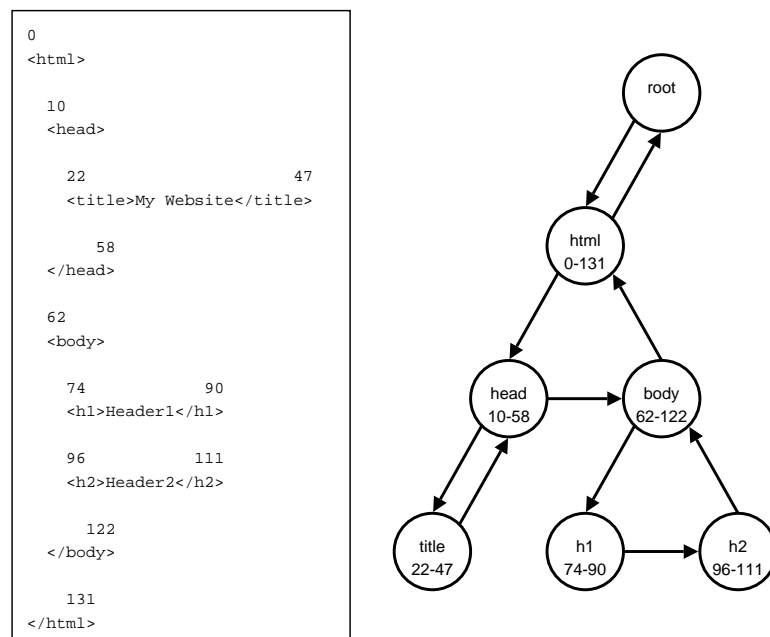


Figure 6.4: An example of how the well-formed sub-trees of an XML document will be stored in the Interval Tree data structure.

- If  $I_2$  should be inserted as right-sibling of  $I_1$  and  $I_1$  already has a right-sibling, then  $I_2$  is passed on to the right-sibling of  $I_1$ , where it is inserted according to these rules.
- If  $I_2$  contains  $I_1$ , then  $I_1$  is converted to a left-child of  $I_2$  and the pointers are corrected.
- If  $I_2$  comes before  $I_1$ , then  $I_1$  is converted to a right-sibling of  $I_2$  and the pointers are corrected.

## Insertion

The insertion operation is used each time an interval is found during checking and needs to be stored. A reduced version of the pseudo-code for the operation is found in **Listing 6.1** (pseudo-code is written in Java/C# style). All the tedious pointer maintenance is left out.

`insertHere` is a node in the tree where an interval can be stored. `INSERT` is always called with the root node as `insertHere` parameter so that `INSERT` always starts from the root of the tree and navigates down the tree to find the correct position to store the given interval. Inserting a malformed or overlapping interval makes `INSERT` return false and an error should be raised. On insertion-success it returns true.

```
1  INSERT(insertHere , inputBegin , inputEnd)
   if ( insertHere.isEmpty() )
   {
5     // If the insertHere-interval is empty
       then insert into insertHere
       return true;
   }
   else if ( inputBegin > insertHere.Begin &&
            inputEnd < insertHere.End )
   {
10    // If the input interval is included in
        the insertHere-interval then insert in
        left-child
       return INSERT(insertHere.LeftChild ,
                     inputBegin , inputEnd);
   }
   else if ( inputBegin > insertHere.End )
   {
15    // If the input interval comes after the
        insertHere-interval then insert in
        right-sibling
       return INSERT(insertHere.RightSibling ,
                     inputBegin , inputEnd);
   }
   else if ( inputBegin < insertHere.Begin &&
            inputEnd > insertHere.End )
   {
20    // If the input interval includes the
        insertHere-interval then insert in a
        new interval and merge it into the
        tree as the parent of the insertHere-
        interval
       return true;
   }
   else if ( inputEnd < insertHere.Begin )
   {
25    // If the input interval comes before the
        insertHere-interval then insert in a
        new interval and merge it into the
        tree as the left-sibling of the
        insertHere-interval
       return true;
   }
   else
   {
30    return false;
   }
```

Listing 6.1: Pseudo-code for INSERT.

## Other Operations

Apart from insertion the other main operations are:

- `SEARCH(begin, end):pointer_to_interval`  
Given a begin and an end position it returns a pointer to the interval having these positions.
- `DELETE(pointer_to_interval)`  
Given a pointer to an interval it removes that interval from the tree.
- `DELETE-INCLUDING-CHILDREN(pointer_to_interval)`  
Given a pointer to an interval it removes that interval and all its children from the tree.
- `DELETE-CONTAINING(position)`  
Given a position it deletes all the intervals containing this position.
- `DELETE-CONTAINED-IN(begin, end)`  
Given a begin and an end position it deletes all intervals that have begin and end position between the given positions.
- `DELETE-OVERLAPPING-NOT-INCLUDING(begin, end)`  
Given a begin and an end position it deletes all intervals that include one of the given position but not both.
- `DELETE-OVERLAPPING-OR-INCLUDING(begin, end)`  
Given a begin and an end position it deletes all intervals that include one or both of the given position.
- `EXPAND(position, amount)`  
Given a position and an amount it does the following:
  1. The length of all intervals including the given position is increased by the given amount.
  2. The positions of all intervals that comes after the given position are increased by the given amount.
- `CONTRACT(position, amount)`  
Given a position and an amount it does the following:
  1. The length of all intervals including the given position is decreased by the given amount. If the length of any intervals becomes zero or less then those intervals are deleted.
  2. The positions of all intervals that comes after the given position are decreased by the given amount. If the begin position of any of the affected intervals becomes less than the given position then those intervals are deleted.

The relevance of these operations will not be explained here but should become clear when they are applied later in this chapter.

## Running Times

In general, the Interval Tree cannot be explicitly balanced by redistributing the nodes because it has to observe the earlier stated insertion and maintenance rules. The mere structure of the tree is the key to implement the above stated operations efficiently.

The running times are as follows, where  $n$  is the number of node in the tree and  $k$  is the number of nodes affected by the actual operation:

- SEARCH:
  - Best case:  $O(\log(n))$ . This occurs if all nodes have two children (binary tree).
  - Worst case:  $O(n)$ . This occurs when the XML document is extremely flat or extremely steep so that the tree is only siblings of siblings or only children of children respectively.
  - Average case: Depends entirely on the structure of the XML document, but something inbetween best and worst case.
- INSERT: Find the place to insert and do the insertion. That is, SEARCH +  $O(1)$ .
- DELETE:  $O(1)$ . Just change a few pointers to remove the node.
- DELETE-INCLUDING-CHILDREN: Also  $O(1)$ .
- DELETE-CONTAINING: Approximately like SEARCH +  $O(k)$ .
- DELETE-CONTAINED-IN: Approximately like SEARCH +  $O(k)$ .
- DELETE-OVERLAPPING-NOT-INCLUDING: Approximately like SEARCH +  $O(k)$ .
- DELETE-OVERLAPPING-OR-INCLUDING: Approximately like SEARCH +  $O(k)$ .
- EXPAND: Worst case:  $O(n)$ . But if intervals are stored as offset and length, where offset is the start position of the current interval minus the start position of the parent-interval, then this operation can be improved to the running time of SEARCH.
- CONTRACT: Like EXPAND

Arguing for the running times of these operations would require to step through their respective algorithms, which will not be done here. Nevertheless, all operations depend upon SEARCH to find the place to start the operation and the running time of SEARCH is fairly easy to grasp.

All the worst case linear running times may not look too impressive, but it is not as bad as it seems. Due to the design of the Interval Tree, the nodes near the root are often the most important because they are the biggest. Actually, all intervals that are included in another interval (i.e. that have a parent) are never used to skip a part of the document because their respective parents are bigger than them and therefore better to use when skipping a part of the document. An interval for a smaller sub-tree is only kept to help in the situation when one of its ancestors are deleted. The children of the deleted interval will now be used to skip parts that otherwise had to be checked again.

## Usage

Apart from storing the newly found well-formed intervals the Interval Tree will mainly be used when running through the document to skip the already checked parts. When doing this the Interval Tree will be traversed in order of interval begin position, but irrelevant sub intervals (intervals which are contained in others) will not be visited. This means that mainly right-sibling pointers will be followed, which makes the traversal similar to running through a linked list.

When an XML document is edited the changes will be merged into the Interval Tree by the `EXPAND` and `CONTRACT` operations. If the sum of the change is that something is added to the document then it needs to be expanded at that point. If something was removed from the document it has to be contracted.

Further usage of the Interval Tree will be explained together with the overall procedure of the optimized well-formedness functional component.

### 6.2.2 Trimming the Data Structure

On large XML documents the Interval Tree storing the well-formed sub-trees may grow too large to be stored in main memory. And long before that it will grow too big to be operated efficiently. So here is a trade off: On the one hand, the more intervals that are stored the better coverage of the well-formed parts and the less time has to be spent re-checking the document. On the other hand, the less intervals that are stored the less time will be wasted on operating the data structure.

The size of the Interval Tree should therefore be adjustable through configuration of the plug-in, so that experiments can be carried out to find the optimal size.

The bottomline line is, though, that the size needs to be controlled. This is done by trimming the tree whenever it grows too big. The trimming consists of deleting the most dispensable intervals. As the purpose of the optimized checker is to skip as much of the document as possible when checking then the bigger intervals are the better. Thus, the smaller intervals are the least significant they are the ones to be deleted first. This can be done by continuing to delete the smallest interval in the tree until the desired size of the tree has been reached.

### Merging Intervals

In some XML documents, a large part of the document can be made up of numerous elements at the same level (with the same parent). This will give a long list of presumably small intervals that will be thrown away during the trimming process even though they together cover a large part of the document. And if the entire document is one long list of adjacent intervals and there is only space to store half of them, then half the document has to be checked each time a little part of the document is changed.

To avoid this, the trimming process will try to merge the smallest interval with an

adjacent interval before deleting it. If it can be merged then there is one interval less in the Interval Tree (thus the size is decreasing) and the coverage of the file is the same.

Merging two intervals is a new operation on the Interval Tree that has not been listed together with the others:

- `MERGE(pointer_to_interval)`  
If the pointed to interval has a right-sibling and the two intervals are mergeable then they are merge. Else any left-sibling is found and if mergeable then merge.

Two intervals being mergeable requires that they are adjacent in a way that no XML elements are between the two intervals – that is, between the end position of the first interval and the begin position of the second interval. However, XML comments and white space is allowed between them.

Discovering this operation of merging, one might consider to always merge all possible intervals to save space while preserving the coverage of the document. However, this can be just as bad as not merging at all because if all intervals are merged, then one might end up with a few intervals covering the whole document. If then the document is changed and an interval has to be deleted, then the deleted interval will leave a very large part of the document to be re-checked and thus slow down performance drastically.

Thus, only merging as many intervals as required by the trimming process will leave a balanced amount of intervals in the Interval Tree that are as uniformly distributed as possible.

### 6.2.3 The Optimized Checker

The overall procedure for the optimized Well-formedness Checker can be seen in **Figure 6.5**. The diagram is not specific about how the skipping of document parts, checking of XML and storing of intervals is carried out. Instead, it illustrates the process of starting and stopping the checking process, handling changes in the document and reporting the status of the document to the user.

In the upper left corner of **Figure 6.5** there is a procedure-header for `CHECKER`. This is where the procedure starts. Initially, the well-formedness status of the document is set to unknown by the `Well-formedUnknown-message`.

Reception of the message `DocumentChanged` is shown in the upper right corner of the diagram. This occurs when AXE sends a message telling that the document has been changed. Consequently, an internal variable called “document changed” is set to `TRUE`. This is an interrupting event that can happen at all times, which is why it is put in a grayed out box.

The rest of the diagram shows the main looping procedure of the checker. In the beginning of each loop it is checked whether the document has been changed by checking the “document changed” variable. If it has, the document status is set to unknown and

a special procedure for handling changes in the document is called. This procedure will be explained later.

If the document has not been changed the checker proceeds to check if the well-formedness status is unknown. If it is known then the checker should not check the document and will loop back to check for changes in the document.

When the status is unknown the loop will continuously check the next piece of XML that needs to be checked. Continuous checks of the next element in the document will eventually lead to that the whole document has been checked. If the checker finishes without finding any errors the document status is changed to well-formed. If an error is found the document status is set to “not well-formed”.

From the diagram it seems as the checker keeps looping and never stops. This should, however, be implemented as an event-listener that awakens the procedure on document changes and else the procedure should sleep when the the status of the document is known.

### Handling Document Changes

**Figure 6.6** illustrates the procedure being called if the document has been changed. It starts by setting the “document changed” to false variable so that it is able to catch a new document change while handling the current one.

Next, either the Interval Tree operation `EXPAND` or `CONTRACT` is called to adjust the positions of the intervals coming after the change.

The change in the document is reported as a `Change-Interval`, which is an interval that covers the changed part of the document. If any intervals stored in the Interval Tree overlap the `Change-Interval` then they are deleted by the call to the Interval Tree operation `DELETE-OVERLAPPING-NOT-INCLUDING`. This is done because it is uncertain whether they are still well-formed. Instead of deleting them one could have checked them again, but this is unnecessary because they, after being deleted, will be found again when the checker later on checks an interval that includes the `Change-Interval` and therefore also the deleted intervals.

The next two steps are `Load Interval A` and `Is Change-Interval in Interval A?`. If interval `A` is unset then nothing is loaded and the `Is Change-Interval in Interval A?` answers no. If interval `A` is set then it contains the interval in which the previous `Change-Interval` was included. The reason for storing and reloading this interval is found in the following assumption of use:

When a user edits a document it is likely that he will continue to edit the same part of the document until it is well-formed. If the edited part of the document was well-formed before the user started editing it then it is beneficial for the optimized checker not to throw that interval away as soon as an error is found in it because the user might correct that error before moving on to other parts of the document. Keeping the smallest interval

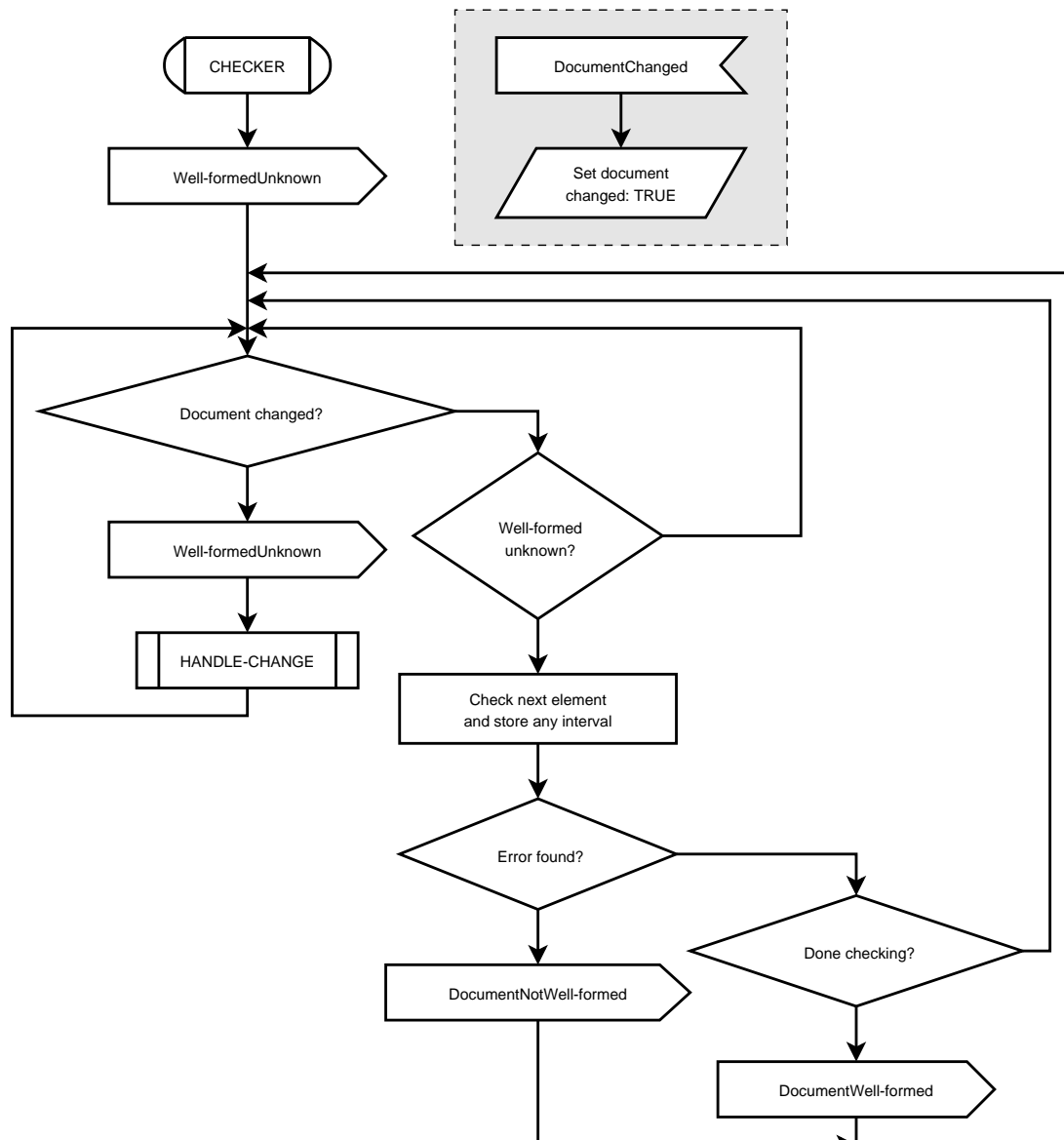


Figure 6.5: An SDL diagram of the procedure for the optimized checker.

including the edited part will thus allow the checker to efficiently recheck that interval on each document change. As soon as the interval is found well-formed interval A is cleared as shown in the bottom right corner of **Figure 6.6**.

However, if the user moves on to another part of the document before the previously edited part was well-formed then interval A and any intervals that overlap or include interval A have to be deleted because they are no longer well-formed. This is done by a call to the Interval Tree operation `DELETE-OVERLAPPING-OR-INCLUDING`.

After that, the checker will want to find the smallest interval that includes `Change-Interval`. This interval will naturally be the smallest part of the document to recheck to see if the changed part is well-formed.

If an interval is found it is stored as interval A and the procedure goes to the checking-loop. If an interval is not found then the `HANDLE-CHANGE` procedure return back to the above described `CHECKER` procedure. Then the `CHECKER` will then check the whole document and thereby also the changed part.

In beginning of the checking-loop it is checked whether the document has been changed since the `HANDLE-CHANGE` procedure started. If so, the procedure restarts, which is showed by the arrow going from `Document changed?` back to start in **Figure 6.6**.

The rest of the checking-loop works pretty much as the one in the `CHECKER` procedure. When the loop exits, either with or without an error, it returns to `CHECKER`. If an error is found the status of the document is changed to “not well-formed”.

The only thing missing in the diagrams **Figure 6.5** and **Figure 6.6** is that if the document is well-formed before a document change and the `HANDLE-CHANGE` procedure finds the change to be well-formed then the document status is again set to well-formed. This detail is left out because keeping track of it would make the diagrams more messy.

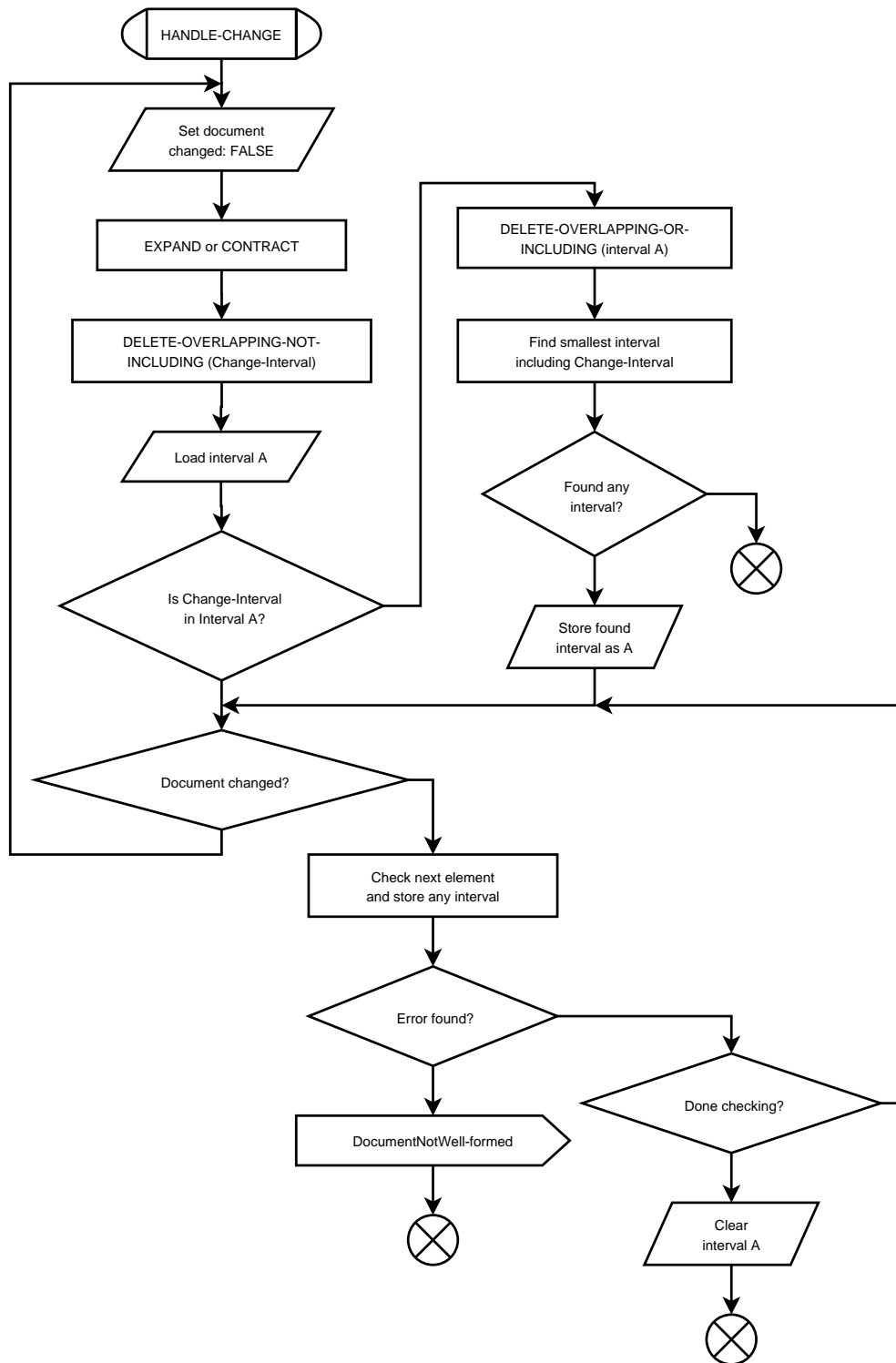


Figure 6.6: An SDL diagram of the procedure for handling changes in the document.

## 6.3 Implementation

The optimized Well-formedness Checker could not be entirely implemented and therefore not tested as a whole because it depends on receiving the “DocumentChanged” message from AXE. This message is never sent because the plug-in providing the message is not finished and not working. This plug-in should have been developed by another group.

Almost all of the optimized Well-formedness Checker is implemented and working but the finishing touch is missing. And so is the documentation of the implementation. Though, here is a short status of the implementation:

- The Interval Tree is implemented in a class called `IntervalTree` and tested to work as described in **Section 6.2 - Design**
- The Microsoft .NET `System.Xml.XmlTextReader` is utilized in a class `Checker` to check XML documents and enabled to skip well-formed parts of the document using the Interval Tree.
- The `XmlTextReader` reads from a `System.IO.Stream` and skipping parts of the document is done by implementing a new stream that feeds the `XmlTextReader` with a contracted version of the XML document that does not contain the well-formed parts stored in the Interval Tree. This is implemented in a class called `ContractedStream` that derives from `System.IO.Stream`. This is tested and works.
- The position of the well-formed intervals has to be located in the stream and this is implemented in a class called `XmlStreamPosition`. It is tested and works for ASCII text, not for some Unicode. The `XmlStreamPosition` uses a buffer implemented in the `ContractedStream` to search for the start tags and end tags of the XML document.

Three things are missing to be implemented:

- The procedure for trimming the size of the Interval Tree is not implemented. It would, however, probably be implemented as putting all the intervals in a priority queue data structure and then keep popping the smallest interval and merge or delete it until the desired size of the Interval Tree is reached. The operation of merging intervals is implemented in the `IntervalTree`.
- The `HANDLE-CHANGE` procedure explained in **Section 6.2 - Design** needs to be implemented. This has not been prioritized due to the missing “DocumentChanged” message.
- The whole Functional component of the optimized Well-formedness Checker needs to be integrated with the User Interface and Controller components so that it will work together with AXE.



# Chapter 7

## Testing

### 7.1 Conditions

To get both a stable development and testing platform the group decided to lock a build of AXE. This resulted in the bugs found in this build will have to be worked around. The group decided to use the build from the third of December as this build contained most of the functionality needed to get the plug-ins working. Locking the build influenced the testing process since known bugs would have to be worked around.

### 7.2 Common Component

The common components are abstract and where hence tested in the following sections through the classes inheriting from it.

## 7.3 Well-formedness Plug-in

The well-formedness plug-in consists of 3 sub-components which due do the different behavior and responsibility have been tested in different manners.

### 7.3.1 Well-formedness Control-Component

The control components of the well-formedness checker where tested by adding a series of debug code parts in the code. In Microsoft Visual Studio .NET it is possible to add these in the code and not compile them in for the finished product. This is done as in **Listing 7.1**.

```
1 #define _debug // if this is left in the top of
   the code the debug code will be compiled
   [snip]
   #if (_debug) // debug code start
       System.Windows.MessageBox.show( ' 'Debug
           testing ' ');
5 #endif // debug code end
```

Listing 7.1: Debug code in VS.net

This tests code coverage and order.

### 7.3.2 Well-formedness User Interface

The user interface of the plug-in (WellformedGUI) has been tested and developed outside the actual plug-in by adding the component to a stand-alone program. The class has been tested by creating a set of status objects and monitoring if the icon displayed corresponded with the content of the status object.

In the same manner the tool-tip and pop up message were tested by supplying the WellformedGUI constructor with error objects containing different description and it was verified that tool-tip and pop up messages contained the correct information.

Additionally it was verified that the tool-tip was functional with different configurations of status and error objects.

### 7.3.3 Well-formedness Functional Component

The functional component tested by opening a series of XML files and observing the result. It was known prior if the XML file was either wellformed or not wellformed and where the error(s) was located. The test was considered a success if the well-formed checker reported errors on the correct places. All the tests run this way were a success.

## 7.4 Validation Plug-in

The validation plug-in like the well-formedness plug-in consists of 3 sub-components and these have like the well-formedness components been tested separately.

### 7.4.1 Validation User Interface

The user interface of the validation plug-in consists of several classes, and these have been tested according to the functional complexity and importance of the classes.

The `ValidationError` class is chosen for extensive testing since it is the class responsible for the actual representation of the individual errors to the user.

The following aspects of the class are tested

- The caption of the graphical representation is consistent with the title of the corresponding error object.
- The description of the graphical representation is consistent with the content of the corresponding error object.
- The appropriate icon is displayed according to the error title (warning/error)
- The correct message is sent to the message queue when the user requests an error/warning to be marked.

### 7.4.2 Validation Functional Component

The validation process is implemented to run as a stand alone program. When the functionality of the validation plug-in reaches a stable state it is to be integrated with the platform. The first implementation of the validation use the validating reader in the Microsoft .NET Framework XML package. For testing the functionality of the validation process XML-documents with known validation errors is used. Then if any unexpected errors occurs, it can be concluded that bug/bugs was is likely to originate in the validation code.

### 7.4.3 Validation Control-Component

When the validation plug-in is integrated with the platform. The XML-documents used to test the functional component is once again used. The use of the same XML-documents guaranties that bugs from the actual validation functionality code do not need to be considered. All other bugs must with a high probability come form the integration with the controller.

## 7.5 Acceptance Test

The motivation for the acceptance test is to determine to which extend the the Plug-in Collection satisfies the requirement described in the functional requirements specification, see **Appendix A.3.5 - Validation**.

The acceptance test consists of a step by step walk through of each plug-in, the Well-formedness Checker and Validator, in the Plug-in Collection with focus on the plug-ins' ability to check for well-formedness and validity respectively of the active document.

Each of the two test cases must be carried out with no deviation from the specified procedures. After each step it is observe if any abnormal error or warning messages are given by the platform or the plug-in in question. Each steps written in **bold** contains direct questions. The answer to these questions must be written on the test-table together with descriptions of any abnormal behavior. The result from the running of the acceptance tests is here **Table 7.1** and **Table 7.2**.

### 7.5.1 Acceptance Test of the Well-formedness Plug-in

#### Preconditions

- A running AXE platform.
- Validation plug-in.
- A running version of the XML Editor plug-in.
- A message queue monitor plug-in.
- The file “testDTD.xml” as seen in **Appendix E - testDTD.xml** .

#### Testing Procedure

1. **Load Well-formedness plug-in, did this action succeed without any errors?**
2. Test the plug-ins ability to check a well-formed document.
  - (a) Open the XML document “testDTD.xml” (which is well-formed).
  - (b) **The status icon should now change briefly to gray (may not be observable), then green - representing well-formed, does this happen?**
  - (c) **Introduce a well-formed error by removing a “<” character from the “<body>” tag using the XML Editor. This should make the status change briefly to gray (may not be observable), then red - representing non-well-formedness, does this happen?**
  - (d) **Hold the mouse-pointer over the status icon. Does a tool-tip appear?**

- (e) **Double-click on the status icon. Does a pop-up message explaining the error appear?**
  - (f) **Does the error get marked in the editor view when closing the dialog?**
3. Unload the Well-formedness plug-in and observe that the plug-in shuts down nicely without generating any errors.

## 7.5.2 Acceptance Test of the Validation Plug-in

The scope of this test description is to test the plug-in as regards to its ability to load and unload and the ability to validate an XML-document. Furthermore the ability to display errors using the GUI and using the editor-view.

### Preconditions

- A running AXE platform.
- Validation plug-in.
- A running version of the XML Editor plug-in.
- A message queue monitor plug-in.
- The files “note.dtd”, “note.xsd”, “testDTD.xml” and “testXSD.xml” as seen in **Appendix Appendix C - DTD - Appendix D - XML-Schema - Appendix E - testDTD.xml - Appendix F - testXSD.xml** .

### Testing Procedure

1. **Load Validation plug-in, did this action succeed without any errors?**
2. Testing Validation plug-in using DTD.
  - (a) Open the file testDTD.xml.
  - (b) Start the validator on the open XML document.
  - (c) **The validator should report that the file is valid, did this happen?**
  - (d) introduce a validation error in the document by adding the tag “<banan/>” after “</body>” in the document.
  - (e) Start the validator on the open XML document.
  - (f) **The validator should now display a list of errors in the side-pane, did this occur?**
  - (g) Close the file testDTD.xml.

3. Testing the Validation plug-in using a Schema.
  - (a) Open the file testXSD.xml.
  - (b) Start the validator on the open XML document.
  - (c) **The validator should report that the file is valid, did this happen?**
  - (d) Edit the file document by adding a new element (so that it is still well-formed).
  - (e) Start the validator on the open XML document.
  - (f) **The validator should now display errors in the side pane, did this occur?**
4. Test if the marking of regions are working.
  - (a) **Click on the “< <Mark” button of one of the error-representations, did this mark the error in the appropriate views?**
5. Close the active document.
6. **Unload Validation plug-in, did this action succeed without any errors?**

### 7.5.3 Result of the Acceptance Tests

The acceptance tests was carried out with slight alteration since we did not have access to any version of the XML Editor plug-in. Instead of doing the manipulations described in the acceptance tests a set of files representing the different stages was used.

Here the result from the acceptance test is discussed. The tables filled out during the acceptance test can be seen in **Table 7.1** and **Table 7.2**.

Each of the tables contains a status for each entry in the acceptance test. The status of an entry in the table can be either **Success**, **Failed**, or **NA**. The first two is self explaining the last **NA** meaning that the test entry is not applicable in its current form but a work-around is found.

	Well-formedness plug-in test case.
1.	<b>N/A</b> - It is not possible to explicitly load a plug-in. However the plug-in is unloaded with out errors when the platform is opened.
2.	
a.	<b>Success</b> - But two message boxes appears with the text “This message-box is needed due to a bug in the platform, sorry!” this occurs each time the status icon is changed.
b.	<b>Success</b>
c.	<b>N/A</b> - The XML Editor plug-in does not exist. The effect on the document is simulated by using a normal text-editor and loading the resulting file.
d.	<b>Success.</b>
e.	<b>Success.</b>
f.	<b>N/A</b> - The XML Editor does not exist but the Message: “Plugin.XML.Data.HighLight” is observed on the message-Queue using the Messages plug-in
3.	<b>N/A</b> - It is not possible to explicitly unload a plug-in. However the plug-in is unloaded with out errors when the platform is closed.

Table 7.1: Acceptance Test results for Well-formedness plug-in test case.

	Validation plug-in test case.
1.	<b>N/A</b> - it is not possible to explicitly load a plug-in.
2.	
a.	<b>Success</b> - Two message boxes appears with the text “This messagebox is needed due to a bug in the platform, sorry!”each time the status icon is changed.
b.	<b>Success.</b>
c.	<b>Success.</b>
d.	<b>N/A</b> - The XML Editor does not exist. Instead the error is introduced using a text-editor and opening the generated file.
e.	<b>Success.</b>
f.	<b>Failed</b> - The errors were not displayed in the side pane. They were displayed in a window.
g.	<b>Success.</b>
3.	
a.	<b>Success.</b>
b.	<b>Success.</b>
c.	<b>Success.</b>
d.	<b>N/A</b> - The XML Editor does not exist. Instead the error is introduced using a text-editor and opening the generated file.
e.	<b>Success.</b>
f.	<b>Failed</b> - The errors were not displayed in the side pane. They were displayed in a window.
4.	
a.	<b>N/A</b> - The XML Editor does not exist but the Message: “Plugin.XML.Data.HighLight” is observed on the message-Queue using the Messages plug-in
5.	<b>Success</b>
6.	<b>N/A</b> - It is not possible to explicitly unload a plug-in. However the plug-in is unloaded with out errors when the platform is closed.

Table 7.2: Acceptance Test results for Validation plug-in test case.

# Chapter 8

## Further development

This chapter presents some of the issues in the Plug-in Collection and AXE, where improvements would be in place. Some of the issues can be handled in the group and others needs cooperation between the groups.

### 8.1 Overall System

#### 8.1.1 Collection of Standard Messages in the Platform

One major disadvantage of the platform is that there are no standard collection of messages (for the message queue). A collection of standard messages would make the interaction between plug-ins easier. These standard messages should be concerning

- File operations (open save close ...)
- File manipulations (text added, text removed ...)
- Highlighting functionality between plug-ins (mark errors, mark warnings ...)

Such standard messages would make it easier to develop plug-ins that is compatible with other development teams and no unchecked messages would appear.

#### 8.1.2 Redesign of the Overall System

The quality of the complete system in a whole would be better if there had been another iteration of the design. Most of the misunderstandings between the development groups could be corrected.

### **8.1.3 Better Integration with Under-lying OS**

One thing that would greatly improve the users experience when using the system is a tighter integration with the operative system. This should consist of file registration so that AXE could be the default editor for XML files on the computer.

Another issue that could improve the users experience was a properly designed installation procedure in stead of just supplying the files as a compressed archive.

## **8.2 Common Component**

### **Better Communication with the Editor**

It would have been nice to have tested the Plug-in Collection especially the well-formedness checker with the XML-editor plug-in. This was not possible since we did not have access to any version of the XML-editor during the entire development phase of the project.

## **8.3 Well-formedness Components**

### **Limit CPU-usage**

As it is now the well-formedness checker can use as much as 100% of the CPU. This could be fixed by adding the ability to assign priority to threads, and then giving the well-formedness checker a lower priority.

This would minimize the probability of the well-formed checker taking unnecessary resources.

## **8.4 Validation Components**

### **Better Integration Into the Platform**

Instead of using messages to request text highlighting, using the methods provided by the platform would have been better. Avoiding using a pop-up window to display validation errors would have been better but this since this approach work, a pop-up window has to be used.

### **Using the Interval Tree Approach for the Validation Processes**

We would like to have implemented the validation plug-in by using the Interval Tree as in the well-formed checker. This would have enabled us to make a much quicker vali-

dition control and possibly allowed us to make continues validation of a document.



# Chapter 9

## Process

This chapter contains reflections on the software development processes and tools used through out the project. It discusses the use of a method to guide the analysis and development phases. The management of the project is discussed.

### 9.1 Discussion of the OOAD Model

The discussion is in two different sections Project Phases and Object Oriented Analysis and Design(OOAD) Tools. In the project phases the order of the phases is discussed. In OOAD Tools The tools used, presented in the book [1], are discussed and evaluated.

#### 9.1.1 Project Phases

At the start of the semester the semester-coordinator provided the participating groups with a set of general deadlines. The deadlines was for the completion of the analysis, design, implementation, and test phases. The group tried to follow the deadlines but ran into trouble after the analysis deadline.

The reason the group began to deviate from the deadlines was that the group had limited technical knowledge in how to work with large XML files. To overcome this problem and learn about XML the group started to prototype.

This limited the testing phase disregarding the high priority given to the Testability Quality Attribute.

#### 9.1.2 OOAD Tools

The analysis and design of the problem domain, was made from proposed diagrams in [1]. The structure of the analysis and design chapters was made using the analysis and design document templates in [1].

From the start some adaptation of the methods presented in [1] was necessary. As it is used for management systems, the high technical content made some of the methods less useful. In addition the model also base analysis and design to be developed with customer interaction. This was not possible, because there are no resources for it, this then resulted in practically no customer interaction. That made it impossible to make proper use-cases, so its how the group expected the use-cases would be.

The overall system definition was produced, by merging four system definition proposals one from each group. This first definition did not contain references to the XML related functional requirements, so at the completion of System Requirement Specification(SRS) a new system definition was made. The new system definition was made in a way to ensure no new functional requirements was added.

The OOAD model normally produce classes, structure and behavior by observing an organizational structure, instead we needed to observe an undocumented technical structure. The lack of information and knowledge made it impossible to make class, component and state diagrams. The group tried the observing approach, where class diagrams, component diagrams, state diagrams and event-tables was produced. The only problem was that the diagrams did not provide the needed perspective. These diagrams was instead produced in the prototyping phase, where the structure of the prototype was the base for the diagrams, then the following development on the prototype was documented by correcting the diagrams.

## 9.2 Management Structure

In the project start it was decided that certain roles and protocols should be used. The groups agreed on following roles and protocols in **Table 9.1**.

Protocols
Code Stanards
CVS and Build Guidelines
Testing
Interfaces and Design
Meetings
Project Team Collaboration Tool
Roles
Project Manager
Build Manager
Test Manager

Table 9.1: This table lists the protocols found here **Appendix B - Protocols** and roles.

The selected roles and protocols, roles was made to split work and responsibilities, the protocols was made to give uniform inter group documents and rules on how to inform the other groups of bugs, tests and meetings.

### 9.2.1 Protocols

#### Code Standards

##### Appendix B.1 - Code Standards

The project proposal for this semester suggested that, further development should be done by the following software engineering students. If using the same code standard all code and documentation will appear alike. It should resolve in the groups delivering uniform code and documentation.

The development platform was chosen to be Visual Studio .NET. This decision was made since this IDE would enforce similar indentation and appearance of the control structure between the different developers and thereby make the source more readable.

Further more it was decided that if possible all code should be written in C#, again in order to assure that the source could be read as a whole.

#### CVS and Build Guidelines

**Appendix B.2 - CVS and Build Guidelines** The CVS and Build protocol is a set of rules for how people use CVS and the tasks for build managers. The build managers

started by setting up CVS, they also tried to setup the automatic build process, but it never worked properly. So two members of the s501a group reconfigured the build server to a working state, after-wards the build server produced a nightly build. When errors occurred the group responsible for the part would received an email and the build failed.

## Interfaces and Design

**Appendix B.3 - Interfaces and Design** This protocol was to formulate the process for making interfaces, joined analysis and design documents between the groups. The interfaces was documented using NDoc. This made it easy to uphold the requirements for documenting interfaces, because the NDoc tags followed the required documentations requirements. A negative things is that NDoc is more difficult to navigate through than the more simple, JavaDoc.

The start of the project some inter group analysis and design documents was produced at project manager meetings, only a couple of diagrams, then the production of inter group documents stopped until the project manager in group s503a, took the initiative to make an overall System Requirement Specification(SRS). He started the document and made correction appointed by s501a and our group. The SRS was first completed after it was requested at a week meeting in the end of November, at the week meeting it was decided that a representative from each group, was to form a small group with the responsibility for the completion of the SRS document.

The problem that raise was, that the groups was more focused on getting their own sub-problem to work, than getting all the sub-problems to work as a system. The problem can be a result of having four project managers instead of one project manager with full responsibility. He would then have knowledge of the whole project and by that have a better chance for locating inter group conflicts

## Meetings

**Appendix B.4 - Meetings** All inter group meetings had to follow this meeting protocol, in the beginning this was no particular problem. The problem was that either summaries lacked detail or did not exist, further discussion can be found here **Section 9.2.2 - Management Roles**

### 1. Supervisor Meetings

The supervisor meetings was unstructured, this was our own fault. Agendas was often made a couple of hours before the meeting, and was not emailed to the supervisor prior the meeting. Writing the summaries were often done by individual group members, not by a formally appointed meeting secretary.

### 2. Weekly Project Meetings

The semester coordinator Thomas Vestdam had requested, that all groups meet once a week to discuss project status and possible inter group problems. The

meeting was held in seminar rooms around campus. Soon after these regular meetings was started, groups requested that it should only be necessary to send representatives. This decision made the weekly meetings functioning as project manager meetings.

Another problem with the weekly meeting, was that no specific person had the responsibility for sending out agendas. It resulted in that agendas was more or less random send out by different students, when agendas was sent out prior the meeting.

### **Project Team Collaboration Tool**

**Appendix B.5 - Project Team Collaboration Tool** All Student on the semester agreed on that we needed tools that providing a Calendar, file manager and a bug tracker. Two tools was found that fitted our needs, eGroupWare providing calendar and file manager, Mantis Bug Tracker for reporting and tracking bugs. Both tools are implemented in PHP and are web deployed. There was not any external accessible web-server on the campus net that had support for PHP, so a student provided disk space on an private web-server. The only requirement for client computers was a web browser so no additional software should be installed and all platforms were supported.

- **Mantis Bug Tracker**

The reason why Mantis was chosen, was that a few students have tried it, with a successful result. It also provide a formal formula for writing bug reports and give a list of bugs with status and priority.

One problem with the bug-tracking was that the system was only used in one direction. The platform got a lot of feedback in the form of bugs/feature requests. But no other groups have received any significant reports.

- **eGroupWare**

This was the best candidate of groupware products since it was free and that it provided the functionality needed including calendar and file-manager functionality. One of the only problems with eGroupWare was the procedure of creating users. But this was solved by exporting the Mantis profile table to the eGroupWare profile table.

### **Testing**

**Appendix B.6 - Testing** At the first test manager meeting, the testing protocol was formalized. The tools used for testing and the responsibility was discussed. The tools the groups agreed on using was Nunit, NCover and NAnt. Nunit was used for unit testing. NCover was to test the coverage of executed code, but was never used. NAnt was used to make a testing script, that would make it easier to run tests and provide the ability to customize testing by supplying parameters. The possibility of using NMock was also discussed at the first meeting, but no decision was ever made about its usage.

The responsibility of what to test was also decided, code is not allowed to contain any build errors and all interfaces are thoroughly tested. Testing the code was to be internally controlled in the groups, but still had to follow the requirement of being well tested. The final program had to be put through an acceptance test made by the test managers.

There was not done as much testing, that was planned. The acceptance test was first formalized at the end of the project, normally this is done at the start of a project just after the system requirement specification is finished. The process of coding unit test while implementation the plug-ins never got started. In general a more specific document for testing should have been made, so one would be forced, to consider testing from the start of the project, so coding of unit tests are done while the source are fresh in memory.

## 9.2.2 Management Roles

### Project Manager

**Appendix B.4 - Meetings** The elected project manager in the group was Kristian S. Bødker.

The appointed project managers function as a council. All major decisions, complaints or major project changes are discussed/decided in meetings held by the project managers. These meetings has to follow the meeting protocol.

The experiences made using the meeting protocol, was that the summaries lacked detail. The lack of detail was the reason that decisions was reconsidered, even through no changes was made to the foundation of the decision. It was then introduced, that the project managers should take turn bringing a secretary, so all project managers had full focus on the discussions. The secretaries only role was to provide the not present with detailed summaries.

One big problem that arose during these manager meetings was that the project managers had troubles agreeing on issues that concerned several groups. This resulted in a reluctance to incorporate changes and the cooperation between the four groups suffered.

The semester coordinator requested weekly meetings, where status/problems in the project was discussed, but ended up functioning as project manager meetings.

### Build Manager

The elected build manager in the group was Martin Madsen. The work of the build manager is described in the CVS and build protocol and in **Section 9.2.1 - CVS and Build Guidelines**.

**Test Manager**

The elected test manager in the group was Dan L. Christensen. The work of the test manager is described in the test protocol and in **Section 9.2.1 - Testing**.

### 9.3 Recommendations

The recommendations are what the group think could help, the students that are to have these kind of inter group projects, it helps for the groups on this semester to acknowledge the good and bad things that happened along the project.

It was a good idea by the semester coordinator providing the groups with deadlines. The problem just was that the platform had the same coding deadline as the plug-in groups. It would have helped if dependencies was taken into account, so that platform had to be done a couple of weeks before the plug-ins. this would give the plug-ins groups, because its not realistic for the plug-ins to be that finished the same time as the platform.

The semester coordinator could get more resources, that would enable him to become the project manager, could help in controlling responsibilities between group implementation. The reason why the semester coordinator should take this role instead of a student at the same semester, is that it could be unpleasant for the student when he needs to use authority.

Make a more specified project plan, which then helps the group keeping track of future tasks and the status of running tasks.

During the project the amount of information one had to check each day, was email, calendar, read summaries, bug tracker and do possible extra duties, one felt that this was a project in it self. The quality of summaries was poor, so after reading a summary from a meeting, it was still hard to understand what have been decided. A way to solve this is to reduce the number of summaries to read. Only to write summaries at supervisor and project manager meetings, at the test and build manager meetings could each have a page on a project website where decisions is listed with a description and date. If a new decision have happened it would have been added to the web page.

A problem with testing was, the test manager had problems formulating unit testing to their groups. As soon as the groups started coding, they should start code unit tests. If unit testing are to be performed, it should somehow be forced upon programmers as soon as they start to program.

# Chapter 10

## Conclusion

At the beginning of the project it was decided to develop the following plug-ins to AXE:

- Well-formedness Checker.
- Validator.
- XPath.
- DTD2Schema.

However only Well-formedness Checker and Validator were implemented.

The implementation of DTD2Schema functionality was dropped after the analysis phase. The XPath plug-in was canceled in the start of the implementation phase. The reasons for dropping the DTD2Schema and XPath plug-in were to focus resources on the two most important plug-ins.

The two implemented plug-ins Well-formedness Checker and Validator have been successfully implemented using the same overall architecture and extending some of the same abstract classes.

One major drawback of the Plug-in Collection is that it is not compatible with the newest version of the platform, but this is due to circumstances outside the group. All members of the group feel that the decision to use the build from the 3rd of December was necessary in order to get past the problem of changing platform interfaces. Some group members even think that the decision should have been made even earlier.

The late development of the work assigned to the different groups have resulted in that the Plug-in Collection is not as integrated into the system as we would have liked it to be. The main issues are that the Plug-in Collection do not use the side-pane of the platform and that marking of errors in the XML Editor plug-in is not tested (since no version have been available).

If a multi-project like AXE is to succeed with a more satisfactory result either stronger management is needed or a more separated problem area must be chosen.

The group members agree that we have gained valuable experiences concerning the structure and management of software projects with multiple parties involved. We have also gained experience in applying the OOAD model in a development environment closer to real-life situations than traditionally possible in an educational context.

Another important lesson learned from the project is that although a standardized model is a valuable tool when developing software, one should not force a model upon a project without allowing adaptation to the model.

# Appendix A

## System Requirements Specification

### A.1 Introduction

The purpose of this document is to act as a system requirement specification (SRS) for AXE.

#### A.1.1 Specification Overview

The SRS is organized into the following sections:

- **Introduction**
- **System Overview**  
Provides a brief, high level description of AXE including its definition, goals, objectives and context.
- **Functional Requirements**  
Specifies the functional system requirements of AXE
- **Quality Requirements**  
Specifies the required system quality attributes
- **Design Constraints**  
Documents the design and implementation constraints on AXE

#### A.1.2 Stakeholders

The stakeholders for AXE are:

- Employees at Aalborg University
  - User Representatives, acting user.

- Supervisors, who must approve the students work.
- Students at 5th Semester of Software Engineering
  - **Project Managers**  
Project managers must ensure that the overall architecture and design meet the requirements specified in this SRS.
  - **Developers**  
Developers must implement the requirements specified in this SRS.
  - **Test Managers**  
Test Managers must ensure that the requirements are validated by writing unit tests.
  - **Build Managers**  
Build Managers must ensure that a version control system is set-up, and provide releases.

### A.1.3 References

This specification refers to or complies with the following documents:

- Summary from the PM meeting, 6th October 2004, where the requirements were specified.
- Summary from the meeting following the interview

## A.2 System Overview

This section provides a high level description of AXE including its definition, primary goals, objectives and context.

### A.2.1 Definition

AXE will be an application consisting of a platform and plug-ins. The AXE platform should be a general purpose development platform capable of loading and saving of files of arbitrary size and grouping files into projects. Furthermore it should provide capability for loading and unloading plug-ins. The release of AXE should contain plug-ins that provide support for version control, viewing, validating and editing XML and plain text files.

### A.2.2 Goals and Objectives

The goals and objectives are taken from the project proposal [8].

- Obtain knowledge about central application development techniques, that solves realistic problems.
- Get experience with analyzing, designing, developing and testing an application, which have a base in a complex technical or organizational environment.
- That the project can be released as a sourceforge.net project.

### A.2.3 AXE Context

This documents the context of AXE in terms of the external hardware and software it interacts with.

#### General Context

AXE is divided into three components, the Platform, the Filesystem and the Plug-ins component. Figure A.1 shows how the Filesystem is a sub-component of the Platform and that the Plug-ins component interacts with the Platform. Figure A.1 also shows the two actors that interact with AXE, a user and a programmer. The user interacts with AXE during program usage and the programmer have the possibility to develop new plug-ins (Jigsaw pieces) and add them to the Plug-ins component.

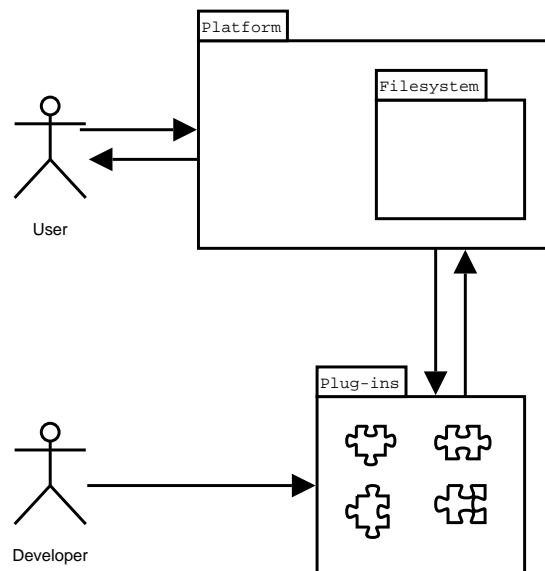


Figure A.1: A rich picture of AXE.

#### External Hardware

The AXE Platform interacts, either directly or indirectly, with the following hardware:

- **Client Hardware**  
Workstations and PCs, which are used by users to perform their tasks.
- **Networks**  
Internet and LAN, which is used for web publishing and connecting to version-control servers, and used to download DTD's / schemas.

### A.2.4 External Software

The AXE Platform interacts, either directly or indirectly, with the following significant software:

- Version-Control Systems, which is the software, running on remote or local machines, that allows the user to manage their files with version control.
- Web Servers, which is the software, running on remote machines, with which the AXE Platform must communicate in order to upload, download or publish documents.
- The .NET Platform, which is the target platform for AXE.

## A.3 Functional Requirements

The functional requirements for the system are described in the following subsections using the following format:

- Name of Requirement
  - **Description**  
This section provides a short description of the requirement.
  - **Priority**  
This describes how important the requirement is to the overall system.

The priority of a requirement can be one of the following:

1. Very Important
2. Important
3. Less Important
4. Irrelevant

### **A.3.1 File-system**

The file-system should make manipulation of large files easier and significantly more efficient by only loading parts of the file into a buffer. It should provide both read and write access to all files and be able to handle several files at once. Large files is in this context files larger than 600 MB following the interview with the acting customer.

Furthermore it should be easy to use the file-system and its advanced features. This is obtained by implementing the same interfaces as the .NET Platforms stream class.

- Priority: Very Important.

### **A.3.2 Multiple Views of Files**

This section documents the requirement that a user shall be able to have different views of a file, if applicable. This requirement is divided into three sub-requirements:

#### **Plain-Text View**

The user shall, as a minimum, be able to view the files as plain-text.

- Priority: Very Important.

#### **XML-Text View**

The user must be able to view XML-files, represented in plain XML syntax. The view shall support color-coding and syntax-highlighting.

- Priority: Important.

#### **Tree View and External Views**

The user should be able to view the files in a more convenient representation depending on the type of the files, i.e. a XML schema could be viewed as a tree or a SVG file as the graphical representation of the file.

The documentation specifying the exact requirements regarding additional views will be prepared by the group working with data views.

- Priority: Important.

### **A.3.3 Version Control**

The version control system should be able to handle a multitude of different file types and version the files according to numerical values.

It should be based on an existing version control server that implements basic features, like checkout, commit, add and ignore. Furthermore the communication with the server should be performed through the editors file handling system. It is based on an existing version control server, so that no time is spent developing a new version control system and the interfaces are well documented.

- Priority: Important.

### **A.3.4 Searching in Files**

The user has to be able to do plain text search in a file. The plain text search should be implemented to perform well in files of arbitrary size. Both time and space complexity should be considered.

- Priority: Important.

### **A.3.5 Validation**

The must be able to check for well-formedness and to validate XML files. The validation process should use DTD and XML Schema.

- Priority: Very Important

### **A.3.6 Platform/Plug-in Structure**

AXE must be developed as platform/plug-in structure. In order to ensure Extensibility and Reuseability the AXE Platform must be developed as a general platform with all file specific functionality implemented as plug-ins.

- Priority: Very Important.

### **A.3.7 User Defined Shortcuts**

Its a requirement that the AXE Platform must be able to handle user defined shortcuts. The AXE Platform shall allow the user to change the shortcuts that control the behavior of the application.

- Priority: Important.

### A.3.8 Customization

The ability to make new plugin-ins for the platform. Add and remove already existing plug-ins from the platform.

- Priority: Very Important.

## A.4 Quality Requirements

This specifies the required system quality attributes. TABLE A.2 show the priorities of the quality attributes in the project. Most of the quality attributes originate from [7].

### A.4.1 Description of Quality Attributes.

Each attribute is described and given a priority that uses the same scale as in the functional requirements. Finally the priorities are argued for.

#### Accessibility

The degree to which the system must be accessible to use by people with disabilities.

- Priority: Irrelevant

The project proposal does not state any demands about accessibility and it was decided to disregard this attribute.

#### Configurability

The degree to which the system should be configurable.

- Priority: Important

This attribute is important because one the demands for this project was to enable support for customizing keyboard shortcuts.

#### Correctness

The degree to which the system ensure that its information is correct.

- Priority: Important

Unit tests are programmed for all functional classes.

**Efficiency**

Determines the efficiency of the resource usage (eg. processor, RAM and disk space).

- Priority: Important

The AXE platform has to be able to work with large files(600 MB) without extensive RAM usage.

**Extensibility**

The extensibility of the application.

- Priority: Very Important

The AXE Platform is extensible by loading plug-ins.

**Interoperability**

The ease with which the application can be integrated with other systems(e.g., legacy applications, databases, etc.).

- Priority: Irrelevant

Irrelevant to the project.

**Maintainability**

The maintainability of the application.

- Priority: Very Important

The application and code has to be well documented, following the project proposal [8].

**Performance**

The performance of the application.

- Priority: Important

The stability when working on large files is more important than performance itself.

**Portability**

How easy it is to port the application to others platforms.

- Priority: Less Important

The application is programmed to run on the .NET platform(CLR). The application are not programmed with portability in mind.

**Reliability**

Reliability is the rate of failures, over a period of time.

- Priority: Important

It is important that failures are kept to a minimum.

**Reusability**

The Reuseability of the application code/program.

- Priority: Very Important

This is very important because the project proposal [8] states Reuseability as a goal.

**Robustness**

The ability of the system to continue to function properly under abnormal circumstances (e.g., invalid inputs, failure of software or hardware components, and failure of applications on which it depends)

- Priority: Important

Unit tests is programmed for most functional methods which ensure some robustness.

**Scalability**

The scalability of the application.

- Priority: Irrelevant

There are no requirements for scalability.

## Security

Securing valuable data, protection from unauthorized use and data integrity.

- Priority: Less Important

Data integrity should be uphold, other aspects are not considered.

## Testability

The testability of the application.

- Priority: Very Important

It is a specific requirement both from the customer and the sw5 project-requirements that the application must be properly tested.

## Usability

Is the design of the user interface.

- Priority: Important

AXE is designed to resemble the look and feel of other Windows applications.

## A.5 Design Constraints

This documents the major architecture and design constraints on the system.

### Programming Language

Required design constraints associated with the use of high-level programming languages.

- The application must be written for the Microsoft .NET platform. Any language that compiles to the .NET platform is acceptable but, in order to ensure consistency across modules, C# should be preferred if possible.
- Where applicable all data should be defined and documented using XML.

## A.6 Tests

Two kinds of tests will be performed, unit tests and acceptance tests.

	Very Important	Important	Less Important	Irrelevant
Accessibility				X
Configurability		X		
Correctness		X		
Efficiency		X		
Extensibility	X			
Interoperability				X
Maintainability	X			
Performance		X		
Portability			X	
Reliability		X		
Reusability	X			
Robustness		X		
Scalability				X
Security			X	
Testability	X			
Usability		X		

Figure A.2: The criterias of the design. Shows how the different quality attributes have been prioritized.

### A.6.1 Unit Tests

Unit tests are required to be performed for all interfaces/function used for interactions between platform and plug-ins. The level of unit testing for functional classes, are set by the individual groups.

### A.6.2 Acceptance Test

The functional requirements from the SRS is the base for the acceptance tests. The tests are only approved by developers and they are individual for each group. A common acceptance test for AXE is not created.



# Appendix B

## Protocols

### B.1 Code Standards

#### Coding style

All project groups must use the coding style used in Visual Studio .NET 2003 for C# code. More information on the coding style can be found at <http://www.csharpfriends.com/Articles/getArticle.aspx?articleID=336>

Special notes are:

- No class is allowed to have public variables, they **MUST** be encapsulated inside a property and then only suitable methods should be accessible (only a get method for read-only variables.)
- Property names and method names must start with a capital letter and then use camel-notation for the remaining text.
- Properties and methods must have descriptive names.

#### Documentation within the source code

Classes are Documented by:

- Writing class summary and usage examples in the start of the file.
- Adding thorough descriptions of each method and public properties.
- Algorithms used inside the code must be stated and referenced to a source of inspiration (e.g. the AALG book with page numbering).

### B.2 CVS and Build Guidelines

It is suggested that everybody uses the same GUI tool. This will promote a common understanding of what is going on, generate common knowledge and make it easier to help each other out on different problems.

The source code should be branched into a branch for each group.

Each group branch is maintained by the responsible group.

Each group is encouraged to define a policy for check-in and check-out. The suggested policy is to check-in code only after review and testing. This way the repository will contain a working revision of the code. Check-in often and keep in sync with the repository. When merge conflicts occur they should be resolved by both the author of the code and the person who recently edited the code.

A group's build manager is responsible for providing a build-ready revision of the branch code at each appointed build event. Since poorly updated branches is a major burden and course of delay on builds, the build managers are encouraged to remind group members of upcoming builds. So they can be carried out on an intact and updated repository. Furthermore to ease the build process the build team can automate the build, this will also help minimize errors in the build.

Only build-managers are allowed to merge the branches into the main trunk and only when they are in agreement.

Before a major build the four branches are merged into the main trunk. As a part of this release process the entire code base must be tagged with an identifier that can help in uniquely identifying the release. The tag should follow this convention  $\gg release_{\{majorversion\#}\}_{\{minorversion\#}\} \ll$ . It is suggested to tag the code first, then check-out the code using the tag and then do the build.

After the build a new branch is created from the main trunk used for debugging the build. The branch is named  $\gg release_{\{majorversion\#}\}_{\{minorversion\#}\}_patches \ll$ . When the patches branch is build again to check the bug fixes, it is not necessary to create a new branch for bug fixing the patch. That is, further bug fixes is done in the same branch.

Each group is encouraged to do minor builds often on their own branch.

Some criteria regarding minimum build regularity and general build manager responsibility (can be revised when the build team is assembled).

- A build of all group branches must take place at least every 2 weeks.
- The build managers team must meet weekly to discuss code status and upcoming builds.
- Build managers are responsible for communicating build deadlines to their individual groups and making sure group branches are ready on build day.

Inspiration was found in: <http://www.tldp.org/REF/CVS-BestPractices/html/index.html>

## B.3 Interfaces and Design

**Interface:**

- There should be a clear and concise description for every parameter of the interface.
- all return values should be explained
- There should be a short description of how the interface is used. Use of examples.
- Every interface should result in a separate document.
- A list of modules/contacts that use this interface should be maintained. with regards to updates.
- When an interface is designed it must be accepted of the other project-groups before being actively used.

**Analysis and Design:**

The overall system should be designed by a small group of people, presumably by the project-managers from each of the 4 groups. The initial design must then be presented to the groups by the project-managers. Each group may now propose changes to this design . (an iterative design process)

- UML!
- Define milestones and uphold them
- The project-managers are responsible for choosing one model which the overall system should follow.
- Each group may choose their own model. (the use of milestones make this possible)
- All design documentation should only reside in digital form.
- All documents should have a common header.

## B.4 Meetings

**Meeting notice:**

- The agenda must be mailed to sw5-6 at least 24 hours before the meeting.
- Topics must be discussed internally in the project groups if the decisions require the complete group's insight.

**Agenda:**

The following are common for all meetings

1. Selection of secretary
2. Selection of meeting moderator
3. Approval of agenda (adding of new items)

**Meeting moderator:**

The meeting moderator is responsible for executing the meeting in a proper manner. Only persons pointed out by the meeting moderator are allowed to speak - in order to have the meeting held quickly and without too much disturbance.

**Secretary:**

Write (and after the meeting rewrite and publish) the meeting results and discussion.

Summary must be published in the appropriate folder on the groupware as well as forwarded to sw5-6 by e-mail at most a day after the meeting.

A summary must contain the following

- Date of the meeting.
- Name of Meeting moderator.
- Name of secretary.
- Invited persons (who actually showed up)
- Items describing the meetings content.
  - Topic
  - Summary of the discussion
  - Decisions

## **B.5 Project Team Collaboration Tool**

**Bug / Issue System:**

- Mantis bug tracker[9]

**Groupware:**

- eGroupWare[10]

## B.6 Testing

### Programs used:

- NUnit (for testing)
- NCover (code coverage)
- NAnt (automated testing)

maybe also NMock ...

### Group testing:

- Testing internally is each groups own responsibility.

### System testing:

- All code submitted to the merge are required to be tested for build errors.
- Interface are required to be thoroughly tested before submitted to the merge.
- acceptance test are made at the next meeting.

Test cases for the overall system are written when the interfaces between groups are settled (around week 43 - see milestones).



# Appendix C

## DTD

The following is a small example of a DTD. The example is taken from [http://www.w3schools.com/dtd/dtd\\_intro.asp](http://www.w3schools.com/dtd/dtd_intro.asp)

It is referred to as the file “note.dtd” in the acceptance test.

```
<!ELEMENT note (to, from, heading, body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
```



# Appendix D

## XML-Schema

The following show a small example of an XML-Schema. The example is taken from [http://www.w3schools.com/schema/schema\\_howto.asp](http://www.w3schools.com/schema/schema_howto.asp)

The content of the file “note.xsd”.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.w3schools.com"
xmlns="http://www.w3schools.com"
elementFormDefault="qualified">

  <xs:element name="note">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="to" type="xs:string"/>
        <xs:element name="from" type="xs:string"/>
        <xs:element name="heading" type="xs:string"/>
        <xs:element name="body" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```



# Appendix E

## testDTD.xml

The content of the file “testDTD.xml”.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE note SYSTEM "note.dtd">
<note>
  <to>Martin</to>
  <from>Dell</from>
  <heading>New Laptop</heading>
  <body>Your new Dell laptop will be delivered
  today, congratulations</body>
</note>
```



# Appendix F

## testXSD.xml

The content of the file “testXSD.xml”.

```
<?xml version="1.0"?>

<note xmlns="http://www.w3schools.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.w3schools.com note.xsd">
  <to>Martin</to>
  <from>Dell</from>
  <heading>Laptop</heading>
  <body>Your new Dell Laptop will be delivered
  today, congratulations</body>
</note>
```



# Bibliography

- [1] Peter Axel Nielsen Lars Mathiassen, Andreas Munk-Madsen. *Objekt Orienteret Analyse og Design*. Marko Aps, Aalborg, 3 edition edition, 2001.
- [2] The World Wide Web Consortium. Extensible markup language (xml) 1.0 (third edition).
- [3] The World Wide Web Consortium. Xml path language (xpath).
- [4] The World Wide Web Consortium. Xml schema part 0: Primer second edition.
- [5] The World Wide Web Consortium. Xml schema part 1: Structures second edition.
- [6] The World Wide Web Consortium. Xml schema part 2: Datatypes second edition.
- [7] Donald Firesmith. Open process framework (opf). Website, 2004. <http://www.donald-firesmith.com/>.
- [8] Thomas Vestdam and Linas Bukauskas. Sw5 projektforslag. Pdf document, 2004. <http://www.cs.auc.dk/Education/SWIng/5/projektforslag.pdf>.
- [9] Mantis Team. Mantis bug tracker.
- [10] eGroupWare Members. egroupware enterprise collaboration tool.