INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen

Oettingenstraße 67 D–80538 München

Ludwig
Maximilians–
Universität
München
LMU

A Toolkit for Advanced XML Browsing Functionalities

Michael Kraus

Diplomarbeit

Beginn der Arbeit:	1. Juli 2000
Abgabe der Arbeit:	20. November 2000
Betreuer:	Prof. Dr. François Bry
	Dr. Norbert Eisinger

Erklärung

Hiermit versichere ich, dass ich diese Diplomarbeit selbständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

München, den 20. November 2000 Michael Kraus

Zusammenfassung

Dieses Projekt besteht aus drei Teilen. Der erste Teil behandelt Themen der Darstellung und des grundlegenden Browsens von Hypertextdokumenten im allgemeinen und von XML-Dokumenten im Besonderen. Ein Hypertextmodell das auf den verschiedenen Modellen der HTML- und XML-Welt basiert, jedoch über diese hinausgeht, wird vorgestellt. Danach werden Probleme, die aufgrund spezieller Eigenschaften dieser vorgegebenen Modelle auftreten, diskutiert.

Im Gegensatz zum ersten Teil, dessen Sicht eines Hypertextmodells sich auf einer niedrigen Ebene befindet, behandelt der zweite Teil das Thema der fortgeschrittenen Browsing-Funktionalitäten. Zwei solcher fortgeschrittenen Browsing-Funktionalitäten werden im Detail vorgestellt, nämlich Browser-Modi und Lesersichten.

In diesem Projekt wurde als dritter Teil ein Werkzeugkasten für fortgeschrittene XML-Browser-Funktionalitäten, genannt *DT Browser*, entwickelt. Der Werkzeugkasten basiert auf den Prinzipien die in den beiden vorangehenden Teilen behandelt wurden. Ziel ist es, den Werkzeugkasten zur Implementierung und zum Testen von verschiedenen fortgeschrittenen Browser-Funktionalitäten benutzbar zu machen. Eine allgemeine Übersicht über den *DT Browser* wird gegeben, allgemeine Prinzipien der Implementierung, das Design jedes Java-Packages sowie Details der Implementierung selbst werden besprochen.

Abstract

This project consists of three parts. The first part describes topics that deal with the display and basic browsing of hypertext documents in general and of XML documents in particular. A hypertext model that is based on the various models that can be found in the HTML and XML world, but goes beyond them, is introduced. Problems that arise due to some special features of those preexisting models are discussed then.

In contrast to the first part, which presents a low-level view of a hypertext model, the second part covers the topic of advanced browsing functionalities. Two such advanced browsing functionalities are described in detail, namely browser modes and reader's views.

Within this project, a browser toolkit for XML, called the *DT Browser*, was conceived and implemented as the third part. The toolkit is based on the principles discussed in the first two parts. The objective is that the toolkit will be suitable to implement and test various advanced browser functionalities. A general overview of the *DT Browser* implementation, general design principles of the implementation, the design of each package of the implementation in Java as well as details of the implementation itself are described.

Contents

1 Introduction **4**
- 1.1 Overview . . . 4
- 1.2 Outline of this Thesis . . . 5

2 Model of Displaying and Basic Browsing **7**
- 2.1 Hyperlink Model and Browsing Model . . . 7
  - 2.1.1 Overview . . . 7
  - 2.1.2 Hyperlink Model: Pure XML Linking Language (P-XLink) 8
  - 2.1.3 Browsing Model: The XSLT approach . . . 9
  - 2.1.4 Browsing Model: The CSS approach . . . 11
  - 2.1.5 Connecting Hyperlink Model and Browsing Model . . . 12
- 2.2 A Generic Model of Rendering and Basic Browsing . . . 13
  - 2.2.1 Parts of a Hypertext Model . . . 13
  - 2.2.2 Alternative Partitions . . . 19
  - 2.2.3 Implementation: Source Document and Sheets . . . 20
  - 2.2.4 Comparison to Existing Standards . . . 22
  - 2.2.5 Sheets: The XSLT approach . . . 23
  - 2.2.6 Sheets: The CSS approach . . . 23
- 2.3 Linkbase Processing . . . 24
  - 2.3.1 Example . . . 24
  - 2.3.2 Virtual Document Linkbase and Browser's Internal Linkbase . . . 25
  - 2.3.3 Processing Modes . . . 26
- 2.4 Document Grouping . . . 27
  - 2.4.1 Examples . . . 27
  - 2.4.2 Virtual Documents . . . 30
  - 2.4.3 Implementation . . . 31
- 2.5 Source Document and Presentation Document . . . 33
  - 2.5.1 Example . . . 33
  - 2.5.2 Limitations of the XSLT Approach . . . 36
  - 2.5.3 Source-Level and Presentation-Level Hyperlinks . . . 37
  - 2.5.4 Rendering . . . 38
  - 2.5.5 Input Events . . . 40

3 Advanced Browsing Functionalities **42**
- 3.1 Browser Modes . . . 42
  - 3.1.1 Hyperlink Browsing Behaviour . . . 42
  - 3.1.2 Implementation: Sheets . . . 43

3.1.3 Generic Browser Mode Model . . . 44
3.1.4 Deficiencies of XSL and CSS . . . 46
3.2 Reader's View . . . 48
3.2.1 Overview . . . 49
3.2.2 Features . . . 49
3.2.3 Implementation: XLink . . . 51
3.2.4 Problems . . . 52

4 Implementation of a Browser Toolkit 55
4.1 Outline of the *DT Browser* Implementation . . . 55
4.1.1 Model of Displaying and Basic Browsing . . . 55
4.1.2 Advanced Browsing Functionalities . . . 56
4.2 General Design Principles . . . 56
4.2.1 Interfaces . . . 57
4.2.2 Factories . . . 60
4.2.3 Packages . . . 65
4.2.4 Exception Handling . . . 66
4.3 Specific Design . . . 71
4.3.1 Browser . . . 71
4.3.2 Documents . . . 72
4.3.3 Reader's View . . . 73
4.3.4 Sheets . . . 73
4.3.5 Actions . . . 75
4.3.6 Messages . . . 79
4.3.7 User Interface . . . 80
4.4 Implementation Details . . . 81
4.4.1 Implementation Language . . . 81
4.4.2 File Structure . . . 82
4.4.3 Third Party Software . . . 83
4.4.4 Tools of the *DT Browser* Implementation . . . 85

Chapter 1

Introduction

This chapter gives a general introduction to this project.

1.1 Overview

Over the time, the contents of the world wide web have become more and more complex. One reason for this behaviour is that the number of people using the world wide web is continuously increasing. These users create new content and new hyperlinks between both new and existing content and thus increase the size and complexity of the world wide web.

Another reason is the development of new technologies that can be used with the world wide web. Java, Javascript, Flash, ActiveX, MP3, the use of databases and many others make today's world wide web much more complex than it has been a few years ago.

Above all, there is the invention of XML and related technologies, which is about to make fundamental changes to the world wide web. In contrast to the monolithic HTML 3.2 approach, which did not include the concept of style sheets yet, XML is not only a markup language, but comes with a set of other languages to describe hyperlinks (XLink), layout (XSL), schemas (XML Schema), queries (XML Query) etc. Each of these languages goes far beyond what is possible with HTML 3.2.

For example, the hyperlink model of XLink, which is the hyperlink language meant to be used with XML, allows the definition of n-ary, bidirectional hyperlinks which can reside in a document separate from all the hyperlink's participating resources. In contrast to that, HTML hyperlinks are binary, unidirectional and have always to be defined together with the hyperlink's starting resource.

Modern style sheet languages such as CSS and XSL allow authors to specify presentation parameters that depend on the output media, for example computer screens or aural rendering. Additionally, these style sheet languages provide complex presentation parameters such as the direction of writing, which differs from script to script.

This new variety of world wide web technologies allows authors to create sophisticated web content. But providing complex hypertext languages is only one part of the work to be done. A problem yet to be solved is to merge this variety of technologies into a single, large hypertext model. The user does not want to display an XML document, a style sheet or a linkbase, but a hypertext document *as a whole*. There is a number of unexplained questions when it comes to combine XML technologies such as hyperlinks and style sheets.

So far, the Dexter Hypertext Model [HS90] from 1990 was used as a reference. In the light of the new technologies, however, it does not seem to be sufficient any more. That's why many authors, such as [Man98b], try to find new hypertext models that meet the requirements of today's world wide web. Within this project, such a hypertext model has also been introduced.

The tool that enables the user to display hypertext documents is the browser. In order to be able to use the newly created complex web contents adequately, browsers have to become more complex as well. It is no longer sufficient to let the user display hypertext documents and traverse simple hyperlinks. In future, browsers will have to provide *advanced* browsing functionalities to the user, such as browser modes or reader's views, which will be discussed within this project.

1.2 Outline of this Thesis

The second chapter describes topics that deal with the display and basic browsing of hypertext documents in general and XML documents in particular. The term *basic browsing* used here covers browsing functionalities that can be found in today's major browsers, such as Netscape Navigator and Microsoft Internet Explorer, in contrast to *advanced browsing* functionalities which are dealt with in the second chapter.

The first two sections introduce a hypertext model that is based on the various models that can be found in the HTML and XML world, but goes beyond them. The remaining sections in the first chapter deal with problems that arise due to some special features of the preexisting models.

In contrast to the second chapter, which presented a low-level view of a hypertext model, the third chapter deals with the topic of *advanced browsing* functionalities. An application-driven view does not distinguish between notions like data, document, layout and others, which belong to *basic browsing*, but specifies its needs in terms like document view, reader's view, adaptivity, user preferences, browser modes and many more, that is, *advanced browsing* functionalities.

The third chapter describes two such advanced browsing functionalities in detail, namely browser modes and reader's views.

Within this project, a browser toolkit for XML, called the *DT Browser*, was conceived and implemented, which is described in the fourth chapter. The toolkit is based on the principles discussed in the previous two chapters. The objective is that the toolkit will be suitable to implement and test various advanced browser functionalities. In addition to the functionalities discussed and implemented within this project, the toolkit is meant to be used and further developed for

new functionalities in various future projects. Because of this future application and extension, main criteria of the implementation are modularization and configurability.

The first section in the fourth chapter gives a general overview of the *DT Browser* implementation. The second section describes general design principles of the implementation. The next section describes the design of each package of the implementation. In the last section, details of the implementation itself are described.

Chapter 2

Model of Displaying and Basic Browsing

This chapter describes topics that deal with the display and basic browsing of hypertext documents in general and XML documents in particular. The term *basic browsing* used here covers browsing functionalities that can be found in today's browsers, in contrast to *advanced browsing* functionalities which are dealt with in the next chapter.

The first two sections introduce a hypertext model that is based on the various models that can be found in the HTML and XML world, but goes beyond them. The remaining sections in this chapter deal with problems that arise due to some special features of those preexisting models.

2.1 Hyperlink Model and Browsing Model

This section suggests a separation of today's hyperlink models into a (pure) hyperlink model and a browsing model. It also shows parallels to the distinction between generic markup language and style sheet language and defines ways to instantiate a browsing model using mechanisms similar to the ones used by style sheets.

2.1.1 Overview

First, let's have a look at an example of a hyperlink defined in the XML Linking Language (XLink) [DMOT00]:

```
<link xlink:type="extended">
  <locator xlink:type="locator" xlink:href="first_line-index.xml
  #xpointer(poem[@title="Daffodils"]/stanza/line)"/>
  <locator xlink:type="locator"
  xlink:href="wordsworth/daffodils.xml"/>
  <arc xlink:type="arc" xlink:show="new"/>
</link>
```

This hyperlink defines:

- a hyperlink between a certain element of an XML document and another XML document
- to open up a new window if the hyperlink is traversed

In general, a hyperlink defines:

- a semantic relationship between a set of data items (the *hyperlink model*)
- the behaviour of the hyperlink on traversal (the *browsing model*)

Note that in the example, both of these models are specified in the same place. The hyperlink model is specified through the semantic attributes `role`, `arcrole` and `title`, the browsing model is specified through the behaviour attributes `show` and `actuate`. These two models are not distinguished in the current XLink specification, that is, hyperlink behaviour is bound to the hyperlinks themselves.

If these two models were specified separately, it would be possible to bind different browsing models to the same hyperlink model. This is especially important for the future, when many data items are likely to be browsed not only using standard-sized screens, but also very small screens like those of mobile phones or other hand-held devices, with which browsing is likely to take new forms. Also, the distinction between these two models contributes to independence between data modeling and data usage and independence between data modeling and presentation devices, which are both key issues in data modeling.

The distinction between hyperlink model and browser model is analogous to the distinction between a generic markup language, whose purpose is to model the logical structure of data items, and a style sheet language, whose purpose is to specify the rendering for a given logical structure. Thus the distinction proposed here is one more step in the separation of various independent concepts of hypertext. In a similar way that a style sheet defines the rendering of data items, there ought to be a *browsing sheet* defining hyperlink behaviour, that is, the browsing model.

2.1.2 Hyperlink Model: Pure XML Linking Language (P-XLink)

As stated in the previous subsection, XLink does not distinguish between hyperlink model and browsing model. Thus two new languages are needed: a (pure) hyperlink language and a browsing model language. It is fairly simple to obtain a hyperlink language:

The Pure XML Linking Language (P-XLink) is defined equally to the XML Linking Language (XLink) without its behaviour attributes (`show` and `actuate`).

In this way, it is no longer possible to define hyperlink behaviour and thus a browsing model within the hyperlink language. With P-XLink, hyperlinks can only be characterized semantically through the semantic attributes `role`,

`arcrole` and `title`. As an example, the values of the XLink behaviour attribute `show` could be semantically characterized like this:

- `new`: hyperlink to new information
- `replace`: hyperlink to alternative descriptions of the current information
- `embed`: hyperlink to supplementary information

The browsing model could then specify the behaviour of these three types of hyperlinks in the same way as in the specification of the XLink behaviour attribute `show` values, for example.

The example from the previous subsection expressed in P-XLink looks like this:

```
<link xlink:type="extended">
  <locator xlink:type="locator" xlink:href="first_line-index.xml
  #xpointer(poem[@title="Daffodils"]/stanza/line)"/>
  <locator xlink:type="locator"
  xlink:href="wordsworth/daffodils.xml"/>
  <arc xlink:type="arc" xlink:title="new information"/>
</link>
```

This hyperlink still defines a semantic relationship between a certain element of an XML document and another XML document. But it no longer defines to open up a new window if the hyperlink is traversed. It simply titles the hyperlink as *new information*.

Note that P-XLink ensures maximum compatibility with the existing XLink standard, which was one of the objectives of this project. Because P-XLink is almost similar to XLink, existing XLink tools can also be used for dealing with P-XLink hyperlinks. It has to be ensured only that no behaviour attributes are used.

2.1.3 Browsing Model: The XSLT approach

As mentioned in the previous subsection, a language for defining a browsing model is needed. Currently two different style sheet languages exist in the XML world: Extensible Stylesheet Language (XSL) and Cascading Style Sheets (CSS). Both of them can be used to define a browsing model as well. This subsection covers the XSL approach, while the next subsection deals with the CSS one.

XSL [ABC+00] consists of two parts:

- a language for transforming XML documents (XSLT) [Cla99b]
- an XML vocabulary for specifying formatting semantics (FO)

Using XSL as style sheet language works as follows: An XML document (*source*) is *transformed* using an XSLT style sheet into an FO document (*target*). Similarly, it can be used as a browsing sheet language:

The source is the definition of a hyperlink model, that is a linkbase specified in P-XLink.

The transformation is carried out using an XSLT browsing sheet. The only difference between an XSLT style sheet and an XSLT browsing sheet is its purpose: An XSLT style sheet specifies the rendering of a class of XML documents whereas an XSLT browsing sheet specifies the behaviour of a set of hyperlinks contained in a linkbase. In both cases, XSLT can be used as transformation language. The two cases differ only in source and target languages considered. Note that although it is possible to carry out complex transformations from source to target using XSLT, only very simple transformations are needed when XSLT is used as a browsing sheet language, as can be seen in the example below.

The target must be a definition of hyperlinks together with their behaviour. The straightforward solution to this requirement is the original XLink language, which serves exactly this purpose.

Given the P-XLink example from the previous subsection, the following excerpt from an XSLT browsing sheet can serve as definition of a browsing model:

```
<xsl:template match="*[@xlink:title='new information']">
  <xsl:copy>
    <xsl:attribute name="xlink:show">new</xsl:attribute>
    <xsl:apply-templates select="@*|node()"/>
  </xsl:copy>
</xsl:template>

<xsl:template match="*[@xlink:title='alternative description']">
  <xsl:copy>
    <xsl:attribute name="xlink:show">replace</xsl:attribute>
    <xsl:apply-templates select="@*|node()"/>
  </xsl:copy>
</xsl:template>

<xsl:template
match="*[@xlink:title='supplementary information']">
  <xsl:copy>
    <xsl:attribute name="xlink:show">embed</xsl:attribute>
    <xsl:apply-templates select="@*|node()"/>
  </xsl:copy>
</xsl:template>
```

As result, we obtain exactly the same document as it was presented in the XLink example at the beginning of this section, that is, the XSLT browsing sheet did define the behaviour of the P-XLink hyperlink. Values for the behaviour attribute `actuate` could be given in a similar way as shown in this example.

2.1.4 Browsing Model: The CSS approach

This subsection explains how to use Cascading Style Sheets (CSS) for defining browsing sheets. Actually, CSS is a simple mechanism for adding style (for example fonts, colors, spacing) to XML documents (amongst others). This is achieved by assigning property/value pairs to document elements. Although the CSS specification [BLLJ98] does not state this explicitly, CSS consist of two parts, just like XSL:

- a generic mechanism for assigning property/value pairs to document elements
- a vocabulary for property/value pairs

The current CSS vocabulary deals (mainly) with style issues, but it could be easily extended to cover browsing models as well. All that is needed are two new properties and their possible values that correspond to the XLink behaviour attributes:

'show'

Value: new|replace|embed|other|none
Initial: UA specific
Applies to: `simple`- and `arc`-type P-XLink elements
Inherited: no
Percentage values: N/A

'actuate'

Value: onLoad|onRequest|other|none
Initial: UA specific
Applies to: `simple`- and `arc`-type P-XLink elements
Inherited: no
Percentage values: N/A
(For more information about the notation of these definitions, see the CSS specification.)

The semantics of these properties are exactly the same as with the XLink behaviour attributes.

The following example will define the same hyperlink behaviour as the XSLT example from the previous subsection:

```
[xlink:title='new information']
{
  show: new
}

[xlink:title='alternative description']
{
```

```
  show: replace
}

[xlink:title='supplementary information']
{
  show: embed
}
```

Note that unlike the XSLT approach, the CSS approach does not make use of the original XLink language, as it does not rely upon the concept of a target language.

2.1.5 Connecting Hyperlink Model and Browsing Model

In addition to defining a hyperlink model and a browsing model for a certain document, it is also necessary to connect these models together. The XLink specification describes how to connect linkbases to XML documents using a hyperlink with a special arc role. Because of the parallels between style sheets and browsing sheets, a new mechanism that is similar to the one for associating style sheets with XML documents [Cla99a] can be used:

Browsing Sheets can be associated with an XML document by using a processing instruction whose target is `dt-browsingsheet` (`dt` for *diploma thesis*).

The following pseudo attributes are defined:

```
href CDATA #REQUIRED
type CDATA #REQUIRED
title CDATA #IMPLIED
media CDATA #IMPLIED
charset CDATA #IMPLIED
alternate (yes|no) "no"
```

The semantics of the pseudo-attributes are exactly the same as with the `xml-stylesheet` processing instruction.

The following example shows an XML document that is associated with a linkbase and a browsing sheet:

```
<?xml version="1.0" encoding="iso-8859-1"?>

<?dt-browsingsheet href="sheets/normal.xsl" type="text/xsl"?>

<document xmlns:xlink="http://www.w3.org/1999/xlink/namespace/">
  <link xlink:type="extended"
  xlink:role="xlink:external-linkset">
    <locator xlink:type="locator"
    xlink:href="sheets/linkbase.xml"/>
  </link>

  <!-- document content to follow -->
</document>
```

Note that the intention is to associate browsing sheets with the XML document, not with the linkbase. Thus different documents can use the same linkbase with different browsing sheets and therefore different hyperlink behaviour.

2.2 A Generic Model of Rendering and Basic Browsing

The previous section identified four separate parts of a hypertext model: data model, hyperlink model, browsing model and presentation model. The instantiations of these models are document, linkbase, browsing sheet and style sheet, respectively. This section reveals other parts of a hypertext model and puts them together with the previously mentioned ones in a generic basic hypertext model. It also shows how these partial models can be instantiated using a generic sheet mechanism that is similar to the one used with style sheets and browsing sheets.

2.2.1 Parts of a Hypertext Model

Data

The following example shows some data represented in XML:

```
<library>
  <author>
    <surname>Keats</surname>
    <forename>John</forename>
    <born>1795</born>
    <died>1821</died>
  </author>

  <author>
    <surname>Shelley</surname>
    <forename>Percy Bysshe</forename>
    <born>1792</born>
    <died>1822</died>
  </author>

  <poem title="Ode to a Nightingale" author="Keats">
    <stanza>
      <line>My heart aches, and a drowsy numbness pains</line>
    </stanza>
  </poem>

  <poem title="Ode to Psyche" author="Keats">
    <stanza>
      <line>O goddess! hear these tuneless numbers, wrung</line>
    </stanza>
```

```
  </poem>

  <poem title="Ode on a Grecian Urn" author="Keats">
    <stanza>
      <line>Thou still unravish'd bride of quietness,</line>
    </stanza>
  </poem>

  <poem title="Remorse" author="Shelley">
    <stanza>
      <line>Away! the moor is dark beneath the moon,</line>
    </stanza>
  </poem>

  <poem title="The Question" author="Shelley">
    <stanza>
      <line>I dream'd that, as I wander'd by the way,</line>
    </stanza>
  </poem>

  <poem title="The Invitation" author="Shelley">
    <stanza>
      <line>Best and brightest, come away!</line>
    </stanza>
  </poem>

  <!-- more authors and poets to follow -->
</library>
```

This data models a library containing a small number of poems by English poets of the Romantic period. For each poem, title, author and first line are given. Also, there's information about each author's name, date of birth and death. The element and attribute names like `library`, `poem`, `title` and `author` all stem from the application domain.

In this section, data is seen in contrast to documents, which are covered below.

Documents

The following is an example of a document, represented again in XML:

```
<page>
  <title>Percy Bysshe Shelley (1792-1822)</title>

  <heading>Index of Titles</heading>

  <list>
    <item>Remorse</item>
    <item>The Invitation</item>
    <item>The Question</item>
```

```
  </list>

  <heading>Index of First Lines</heading>

  <list>
    <item>Away! the moor is dark beneath the moon</item>
    <item>Best and brightest, come away!</item>
    <item>I dream'd that, as I wander'd by the way</item>
  </list>
</page>
```

This document contains a title stating the name of Percy Bysshe Shelley as well as his date of birth and death, two headings and two lists with three items each. The first list shows an index of the titles of Shelley's poems, the second list shows an index of Shelley's poem's first lines.

Note that this document could be generated from the data representation shown above, for example using an XSLT style sheet. However, there are several differences between these two representations:

Namespace The document element and attribute names stem from a different domain than in the data example. They do not model a library, authors and poems. Instead, they describe conceptual parts of a document, namely titles, headings, lists, items etc.

Structure The tree structure in both examples differs fundamentally.

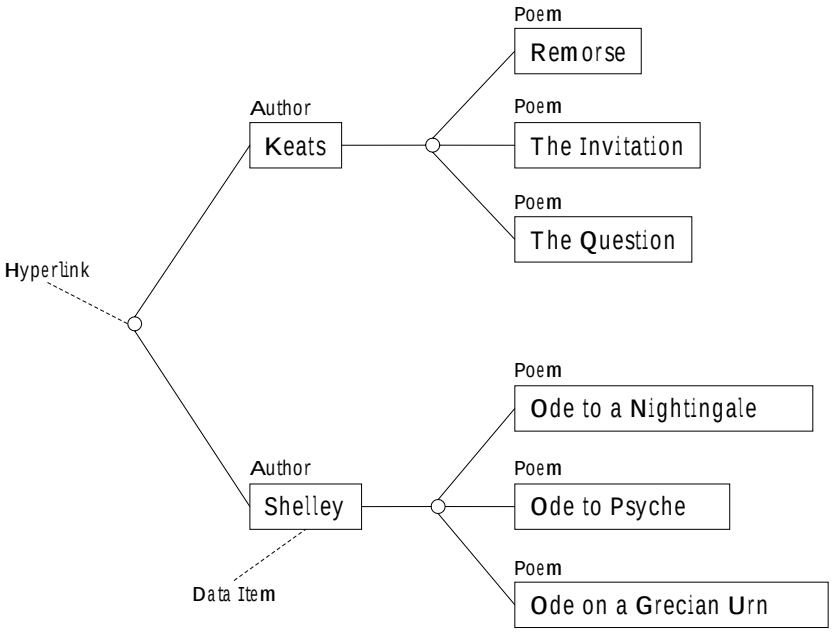
Content While the data example contains information about poems from both Shelley and Keats, the document example contains information about Shelley only. Also, the document shows new content that is not represented in the data example, like the texts "Index of Titles" or "Index of First Lines".

Storage Data like the one from the example is perfectly suited for storage in a database, whereas documents are typically stored in text files or realized as streams sent to a browser. Moreover, there's no need to materialize the document at all: It can be generated from the data and an XSLT style sheet.

Generic markup languages like XML [BPSM00] allow the definition of both data- and document-oriented content.

Note that the term *document* is ambiguous: First, it has the meaning of an XML document. Second, it is used here in contrast to the term *data*. Both data and documents may be represented using XML documents, however. Because both meanings are well-established in their respective fields, they are used here simultaneously.

Figure 2.1: Hyperlink Example



Hyperlinks

As mentioned in the previous section, hyperlinks model semantic relationships between data items. Figure 2.1 shows an example of several data items interconnected by hyperlinks.

The left hyperlink interconnects the two authors, while the two hyperlinks on the right connect each author with all of his poems, respectively.

The features of today's sophisticated hyperlink models exceed those of simple hyperlink models like HTML. The hyperlink model of the XML Linking Language (XLink) allows the following, for example:

- hyperlinks between more than two resources
- hyperlinks can be stored in a location separate from their resources

As a consequence, hyperlinks are no longer meant to be considered only as parts of hypertext that can be traversed by the user, but as metadata about hypertext. Various new applications that exceed the simple traversal of a hyperlink by the user are likely to appear, for example within the field of information retrieval. Therefore, hyperlink behaviour is no longer considered to be a part of the hyperlink model (see below).

Style

Data items may not only be rendered as typewriter text, as it was done so far in this text. Figure 2.2 shows some elements of the document example from above:

Figure 2.2: Style Example

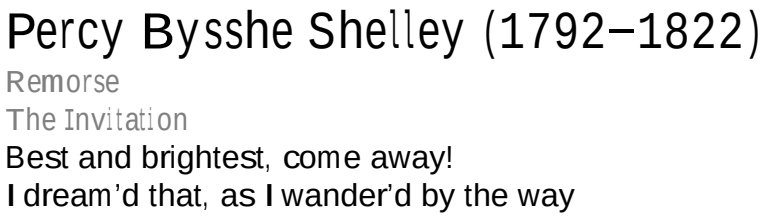
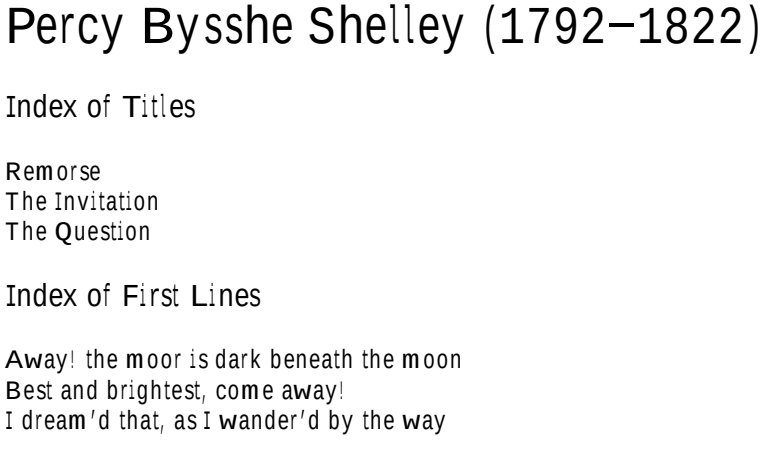


Figure 2.3: Layout Example 1



In this example, the `title` element is rendered in black using the *Times* font with a size of 20pt. The two `item` elements from the first list are rendered in grey using the *Helvetica* font with a size of 12pt. The two `item` elements from the second list are rendered in black using the *Times* font with a size of 12pt.

In the case of visual rendering, the term *style* covers fonts, colors, padding and borders. In this thesis, layout is distinguished from style (see below). Also, issues like automatic numbering of list items, automatic generation of table of contents etc. are not considered to be part of a style model. These issues can be seen as automatic data-to-document or document-to-document transformation.

Common style languages that cover also layout and data/document transformation issues are CSS [BLLJ98], XSL [ABC+00] and DSSSL [ISO].

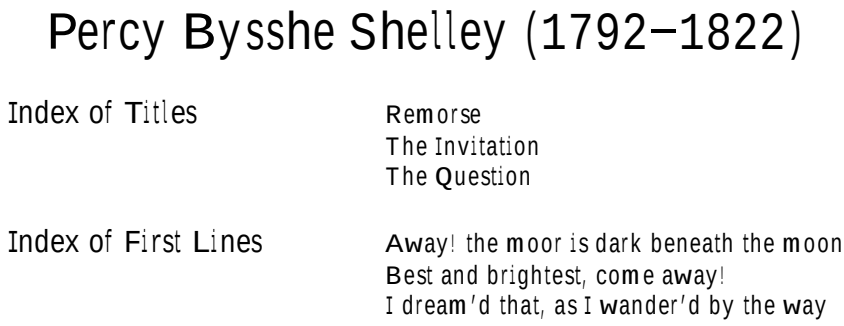
Layout

Figures 2.3 and 2.4 show two different layouts of the same set of elements from the style example:

The first layout stacks all elements (title, two headings and lists) on top of each other, while the second layout places the lists to the right of the headings and centers the title.

As stated above, this thesis distinguishes between *style* and *layout*. The style model defines *how* to render data items, the layout model defines *where* to put them.

Figure 2.4: Layout Example 2



Layout is not always a spatial problem: In the case of aural rendering, for example, the layout model has to decide *when* to render a certain element, that is the order of elements as well as pauses between them.

Layout also has to deal with a varying number of dimensions: Monoaural rendering is one-dimensional, rendering on paper is two-dimensional, but has to deal with pagination issues, and rendering on a computer screen is three-dimensional, as a theoretically unlimited number of windows offers a third dimension.

Browsing

In the previous section, the distinction between hyperlink model and browsing model was illustrated. Here, a more general view of browsing will be presented. In contrast to the *rendering* aspects of a hypertext model, which were described so far, *browsing* introduces dynamic issues.

General Browsing Model: Scripting Languages Generally, each browsing action can be seen as invocation of a piece of program code. For example, if the user traverses a hyperlink, a piece of program code may open a new window and render the target of that hyperlink within it. Or, if the mouse cursor hovers over a certain element, the color of that element may change. In this model, program code that implements browsing functionalities is not considered to be an integral part of the browser, as it is done today, but it is seen as part of the hypertext. This point of view is similar to that of the Document Object Model (DOM) [Woo98].

The most popular scripting language is ECMAScript [ECM99], a standardized version of JavaScript. The following example ECMAScript code changes the color of an HTML element while the mouse cursor is hovering over it:

```
<p>
  I wander'd lonely as a
  <span id="cloud"
    onMouseOver="this.style.setAttribute('color','red','false')"
    onMouseExit="this.style.removeAttribute('color','false')">
    cloud
  </span>
```

```
</p>
```

While the mouse cursor is hovering over the word *cloud*, its color will be changed to red.

Specialized Browsing Models As mentioned above, the term *browsing* covers more than just hyperlink traversal. With the general model of scripting languages described in the previous paragraph, any browsing functionality can be implemented. However, this approach is implementation-dependent. In the case of hyperlink behaviour, for example, it would be much better to use the mechanism of a separate browsing sheet described in the previous section. There, it was called *browsing model*, but within the current context, it should actually be called *hyperlink behaviour model*, as it covers only that issue. In contrast to the scripting language model from above, which allows to specify arbitrary browsing functionalities, the hyperlink behaviour model can be seen as an implementation-independent, specialized, part of a browsing model.

In the same way, it would be possible to outsource other parts of the general scripting language model as well, for example a hover model. Such a hover model would be able to specify the behaviour of data items if the mouse cursor hovered over it. The ECMAScript example from the previous paragraph could then be specified in a way like this:

```
#cloud
{
  hoverColor: red
}
```

This example uses the CSS approach, but an XSLT approach could also be used, as it was shown in the previous section, or any other similar approach.

2.2.2 Alternative Partitions

The division of a hypertext model into the parts data, documents, hyperlinks, style, layout and browsing is not the only possibility.

As it was mentioned above, *style* covers the four separated parts fonts, colors, padding and borders. It would be possible to put each of these four subparts on the same level as the main parts of the hypertext model. Then, the hypertext model would be: data, documents, hyperlinks, fonts, colors, padding, borders, layout and browsing. Also, the browsing model could be divided into scripting languages, hyperlink behaviour and hover behaviour, as described above.

On the other hand, parts that were described to be separate can be combined. So cover common style sheet languages both the issues of style and layout (and other things), for example.

The hypertext model described here is generic in the sense that it allows any number of partial submodels. Not all parts are necessary for all applications, and there might arise new aspects of hypertext in the future, which can easily be incorporated with this model just by defining new partial submodels.

The decision about which concrete separation to choose depends on the following two considerations: If the hyperlink consists of many separate parts, it is possible to define more complex hypertexts, but the definition itself becomes more complex as well. On the other hand, if the hypertext model consists of a small number of parts, or even no parts at all (like HTML 3.2, which does not know about the concept of style sheets), it is much easier to define small hypertexts, as every aspect can be specified in the same place.

2.2.3 Implementation: Source Document and Sheets

So far, the hypertext model has been divided into an arbitrary number of parts. Now, we need a generic method to instantiate these partial models. Also, there must be a method to glue together these instantiations.

Defining Source Document and Sheets

The complete hypertext is divided into two parts. First, there is the content of the hypertext. The content resides in the *source document* and can be represented in both data or document form, as described above. Second, there's a number of *sheets*, one for each remaining part of the hypertext model. Style sheets are alread widely known and the previous section introduced the new concept of browsing sheets. The model described in this section covers document sheets, hyperlink sheets, style sheets, layout sheets and browsing sheets:

- A document sheet defines how to transform data items into document form. This becomes obsolete if the hypertext content is already defined in document form.
- A hyperlink sheet is called linkbase in XLink vocabulary.
- The term *style sheet* used here covers only style issues, as described above. The widely known CSS, XSLT and DSSSL style sheets cover document and layout issues as well.

As a summary, the source document defines the content of the hypertext, while the sheets define its rendering and browsing properties.

The following is an example of a document sheet represented in XSLT:

```
<xsl:template match="library">
<page>
  <title>
    <xsl:value-of select="author/forename"/>
    <xsl:value-of select="' '"/>
    <xsl:value-of select="author/surname"/>
    (<xsl:value-of select="author/born"/>
    <xsl:value-of select="'-'"/>
    <xsl:value-of select="author/died"/>)
  </title>
```

```
  <heading>Index of Titles</heading>

  <list>
    <xsl:for-each select="poem">
      <item><xsl:value-of select="@title"/></item>
    </xsl:for-each>
  </list>

  <heading>Index of First Lines</heading>

  <list>
    <xsl:for-each select="poem">
      <item><xsl:value-of select="stanza/line"/></item>
    </xsl:for-each>
  </list>
</page>
</xsl:template>
```

This example transforms the data example to the document example, both from the beginning of this section.

Connecting Source Document and Sheets

The method of connecting browsing sheets to an XML document described in the previous section can be generalized to allow any sheet to be connected to a source document:

Sheets can be associated with a source document by using a processing instruction. The target of the processing instruction denotes the type of the sheet (`dt` for *diploma thesis*):

```
dt-documentsheet
dt-hyperlinksheet
dt-stylesheet
dt-layoutsheet
dt-browsingsheet
```

Note that `dt-stylesheet` is synonymous to `xml-stylesheet`. The following pseudo attributes are defined:

```
href CDATA #REQUIRED
type CDATA #REQUIRED
title CDATA #IMPLIED
media CDATA #IMPLIED
charset CDATA #IMPLIED
alternate (yes|no) "no"
```

The semantics of the pseudo-attributes are exactly the same as with the `xml-stylesheet` processing instruction.

Table 2.1: Hypertext Model and Existing Standards

	HTML 3.2	HTML 4.0	XML	XLink	CSS	XSL/XSLT
data	X	X	X			
document	X	X	X			X
hyperlinks	X	X		X		
style	X				X	X
layout	X				X	X
browsing	X	X		X		

In the case of hyperlink sheets (linkbases), another approach already exists: The XLink specification describes how to connect linkbases to XML documents using a hyperlink with a special arc role. This could also be extended to a generic mechanism for connecting sheets to a source document by introducing a new arc role for each type of sheet. The following example shows an XML source document that is associated with a hyperlink sheet (linkbase) using the processing instuction mechanism, and with a style sheet using the special arc role mechanism:

```
<?xml version="1.0" encoding="iso-8859-1"?>

<?dt-hyperlinksheet href="sheets/links.xml" type="text/xml"?>

<document xmlns:xlink="http://www.w3.org/1999/xlink/namespace/">
  <link xlink:type="extended"
  xlink:role="dt-stylesheet">
    <locator xlink:type="locator"
    xlink:href="sheets/style.css"/>
  </link>

  <!-- document content to follow -->
</document>
```

The connection between source documents and sheets could also be implemented using more general metadata mechanisms like the Resource Description Framework (RDF) [LS99].

2.2.4 Comparison to Existing Standards

Table 2.1 gives an overview about the connection between the parts of the hypertext model described in this section and a number of existing standards.

Notes on the table:

- XML can be used to model hypertext content in both data and document form. If the content is in data form, it has to be transformed to document form using XSLT, for example.
- XLink, XSL and XSLT themselves are XML as well, but the XML column means *generic XML* only.

- XML itself offers rudimentary hyperlink support through the attribute types `ID/IDREF/IDREFS` and `ENTITY/ENTITIES` as well as through external entities.
- In contrast to XLink, P-XLink is only a hyperlink model, not a browsing model. The previous section showed how both CSS and XSLT can be used to implement a browsing model for P-XLink.
- CSS also offers rudimentary document support through the `:before` and `:after` pseudo-elements.

2.2.5 Sheets: The XSLT approach

Originally, XSLT is meant to be used as a style sheet mechanism. Actually, XSL covers document, style and layout issues. In the previous section, it has been shown how XSLT can be used to implement browsing sheets as well. In fact, this mechanism can be generalized to implement arbitrary sheets.

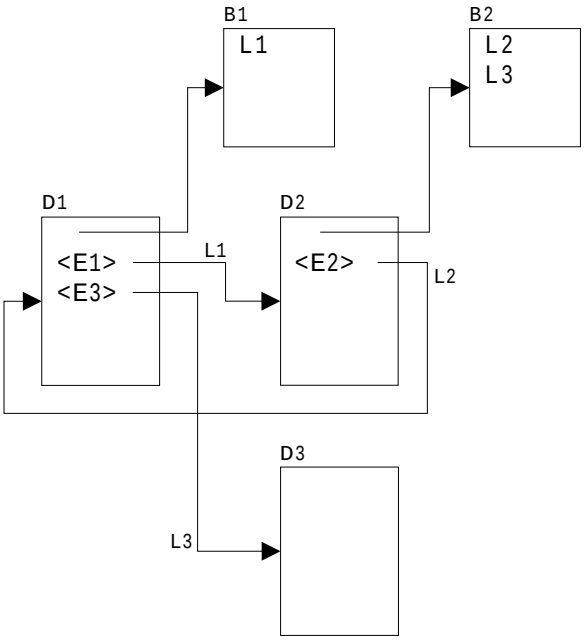
Using XSLT as a language for transforming XML documents implies the concepts of source and target language. For example, if XSLT was used for all parts of the hypertext model except the hyperlink model, which is meant to be implemented in XLink, a source document would be transformed in several steps into a document that is ready to be rendered. Because every step takes the output from the previous one as its input, all target languages must be able to preserve all information that is represented in the source language document. For example, the language used as target of the layout step must also represent information about style, hyperlinks and documents. And the last language in the transformation process must be able to represent information about all parts of the complete hypertext model. This is somewhat counterproductive as the hypertext model was split up for benefits like modularity, independence etc. Also, it is necessary to define an order on the parts of the hypertext model, which is an otherwise unnecessary constraint.

The XSLT approach actually involves a transformation sequence, which takes as input a source document and has a *presentation document* as its output.

2.2.6 Sheets: The CSS approach

CSS is a declarative language used to assign property/value pairs to elements. Currently, CSS includes properties for defining document, style and layout. It could be easily extended to include properties for browsing as well (see previous section). In order to separate between the various concepts, CSS has to be split up into several languages, all with the same syntax, differing only in the set of property/value pairs allowed. Doing it this way, there would be a language called Document-CSS, which contains only properties describing features concerning the document concept, Style-CSS, which contains only properties describing features concerning the style concept, Layout-CSS, with properties describing the layout concept and finally Browsing-CSS, with properties describing the browsing concept only. An advantage of the declarative approach of CSS is that there is no need to define an order on the sheets. It makes no difference if style or layout is defined or applied first to the data. The fact that the CSS syntax

Figure 2.5: Linkbase Example



is not an XML syntax should be seen, in the context discussed here, as a real drawback. Indeed it would be rather immediate to turn the non-XML syntax of CSS into an XML one while preserving the declarativity of the CSS approach.

2.3 Linkbase Processing

This section deals with issues specific to XLink linkbases. It argues that there are several possibilities for a browser to process linkbases, a topic which is not covered in the XLink specification [DMOT00].

2.3.1 Example

Figure 2.5 shows an example of two XML source documents and two XLink linkbases.

The two XML source documents D1 and D2 contain a reference to the XLink linkbases B1 and B2, respectively. Linkbase B1 contains hyperlink L1 which associates element E1 in document D1 with document D2. Linkbase B2 contains two hyperlinks: The first one, L2, associates element E2 in document D2 with document D1. The second one, L3, associates element E3 in document D1 with another XML source document, namely D3. This is possible because participating resources of XLink hyperlinks are not restricted to reside in the document from which the linkbase was referenced. Therefore linkbase B2, which is referenced by document D2 only, can also contain a hyperlink which contains a resource that resides in document D1, which does not reference this linkbase.

Now, what happens if the user browses through the above mentioned documents? Let's assume that the browser is displaying document D1 now and has never visited document D2 before. Because linkbase B1 is referenced by document D1, the browser is able to display hyperlink L1, for example by rendering element E1 in a special color. If the user follows hyperlink L1, the browser displays document D2. As linkbase B2 is referenced by document D2, the browser is able to display hyperlink L2 by rendering element E2 in a special color as well. But now the browser has also knowledge of hyperlink L3, which is contained in linkbase B2 as well. Because none of the participating resources of hyperlink L3 resides in document D2, the browser is not able to display this hyperlink now, however. If the user follows hyperlink L2, the browser will show document D1 again. In contrast to the first visit to this document, the browser now has knowledge about hyperlink L3 as well, which associates element E3 in document D1 with document D3. So, it would be possible to display both hyperlinks L1 and L3 now. The question arises which browser behaviours are possible, meaningful and desirable. Because the XLink specification doesn't say anything about what to do in a case like this, this problem is discussed in detail in the following.

2.3.2 Virtual Document Linkbase and Browser's Internal Linkbase

The example given above doesn't cover all possible cases, of course. Therefore an abstract model of documents and linkbases is constructed here.

There are three possible places where an XML source document can define hyperlinks. First, there may be outbound hyperlinks contained in the document itself. Second, the document can contain references to one or more external linkbases which contain hyperlinks. Third, these linkbases may contain references to other linkbases, which contain hyperlinks as well. Note that according to the XLink specification it is possible to constrain the browser to follow recursive linkbase references only up to a certain maximum depth. Therefore it is possible for two different browsers to work with a different set of hyperlinks while processing the same document.

The union of these three sets of hyperlinks (outbound hyperlinks, hyperlinks from directly referenced linkbases and hyperlinks from indirecly referenced linkbases) is called *virtual document linkbase*. While processing a document, the browser internally constructs such a virtual document linkbase. Note that this virtual document linkbase does not contain hyperlinks which have a participating resource that resides in the document, but which are defined in a linkbase that is not referenced by the document, as in the example given above (hyperlink L3 during the first visit to document D1).

There's another construct that resides internally in the browser, which is the *browser's internal linkbase*. The browser's internal linkbase contains all hyperlinks that the browser has knowledge about or is configured to have knowledge about (see below). All hyperlinks with a participating resource that resides in the document currently being displayed are contained in the browser's internal linkbase.

The browser's internal linkbase is subject to change while the user is browsing. Each time a new document is visited, the browser constructs that document's virtual linkbase. Then the browser's internal linkbase is updated, as the virtual document linkbase may contain hyperlinks that the browser does not yet know about, as in the example given above. There are several possibilities for this update procedure, which will be covered in detail below.

Another explanation of the difference between virtual document linkbase and browser's internal linkbase is as follows: The virtual document linkbase is a *static linkbase*, because it can be constructed only from information that is contained within the document and does not change unless the document itself changes. The browser's internal linkbase is a *dynamic* linkbase, because its contents change with each browsing action according to the processing mode used. The same sequence of browsing actions may result in different dynamic linkbases if different processing modes are used.

2.3.3 Processing Modes

There are several different possibilities for the actual behaviour of the browser's internal linkbase.

First, consider a *closed world* scenario. This may be a set of documents that make up an electronic newspaper, an electronic book or a complex technical documentation, for example. In such a scenario, it is possible to have knowledge about all existing hyperlinks within this closed world, even if not all of them are referenced by the document currently being displayed. It is possible to pre-load this set of all existing hyperlinks into the browser, which will be used as the browser's internal linkbase then. While the user is browsing, the browser's internal linkbase won't change, as there are no hyperlinks referenced by a document that the browser doesn't already know about.

Second, there may be an *open world*, for example the world wide web. There, each newly processed document may add new hyperlinks to the browser's internal linkbase. In this case, the browser's internal linkbase is the union of the virtual document linkbases of all documents the browser has processed so far. This behaviour is similar to the one in the example given at the beginning of this section, that is new hyperlinks can appear when a document is visited again, because documents visited in between may have added new hyperlinks with participating resources that reside in the first document to the browser's internal linkbase. The case described here gives the maximum possible knowledge about existing hyperlinks to the browser.

Third, there is another possible behaviour in the *open world* scenario. In this case, the browser deletes all hyperlinks from its internal linkbase each time a new document is processed. This means that the browser's internal linkbase is always equal to the virtual document linkbase of the document currently being displayed. Then the example given at the beginning of this section would not be possible, because the browser already "forgot" about the existence of hyperlink L3 when document D1 is visited for the second time. This behaviour ensures that a document looks similar to the user each time it is visited.

Finally, there are combinations of the three cases described above. There may be a closed world *embedded* in an open world, for example, an electronic newspaper

may reside within the World Wide Web. In this case, there are links that point from the closed world into the open world and vice versa. Then it is not possible to pre-load all hyperlinks of the closed world into the browser, but still the browser has more knowledge than in the open world case described above. Also, it is possible in an open world as well to pre-load linkbases into the browser. These linkbases do not contain all hyperlinks existing in the open world, but they may contain hyperlinks with no participating resource that resides in the document currently being displayed. The browser should provide means to the user to control these various processing modes.

2.4 Document Grouping

This section describes the concept of grouping together several pages of hypertext to form a *virtual document*. This approach is the solution to context information and scope problems, which are described in two examples.

2.4.1 Examples

Frames

Figure 2.6 shows an example of a hypertext document about English verse.

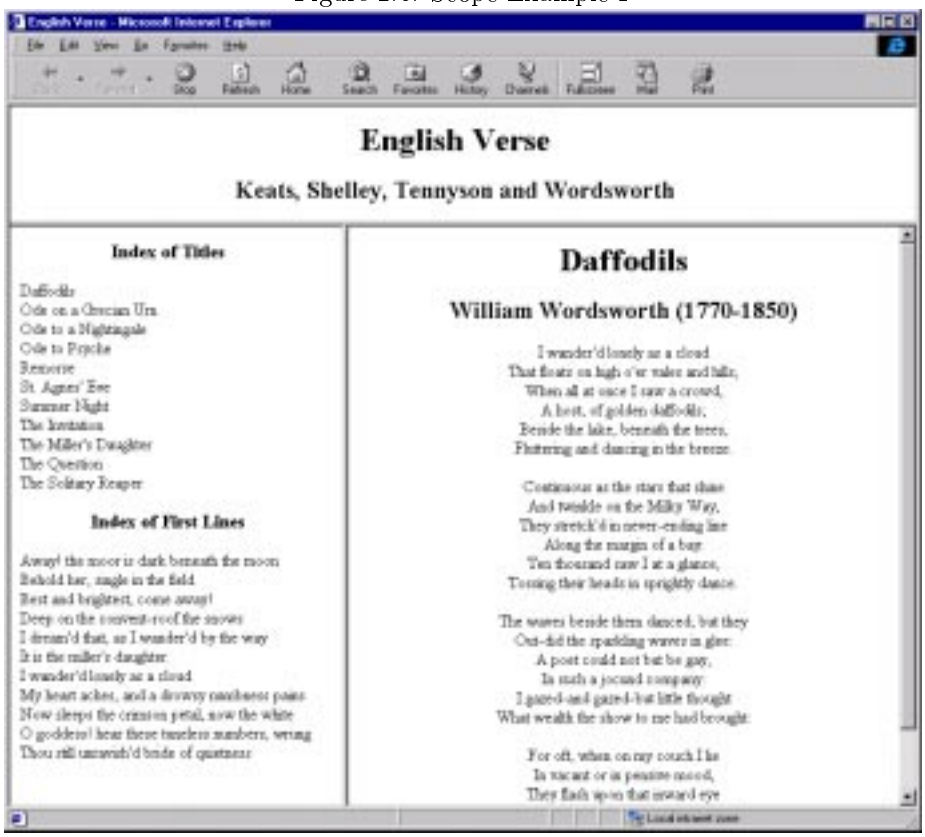
This hypertext document consists of three frames: The frame on the top shows title and subtitle of the page ("English Verse ..."), the frame on the left shows a menu containing two lists — the upper list is an index to the poems' titles, the lower list is an index to the poems's first lines — and the big frame in the center shows a poem itself. If the user clicks on one of the items of the two lists in the frame on the left, the corresponding poem will be shown in the center frame.

Let's assume that the browser shows the document of figure 2.6. The user wants to read the poem being displayed, but he also sees a hyperlink in the left frame to another poem he wants to read afterwards. In order not to forget this second poem, he traverses the hyperlink with the browser option "open hyperlink in new window", an option offered by today's major browsers. But the new window that appears will display the poem only, without the two other frames, as shown in figure 2.7.

This is because the hyperlink has no information about the fact that the poem does not form a page by itself, but is meant to be shown together with the title frame and the menu frame. If our user finished reading the first poem, he closes the window and starts reading the second poem. But this second window misses all the navigation that is provided by the title frame and the menu frame, so the user is unable to further navigate within the set of poems.

As a solution, the browser could attach *context information* to the hyperlink while traversing it and then build the same set of frames in the newly opened window as it existed in the original one. A more general solution will be discussed below.

Figure 2.6: Scope Example 1



Style Sheets

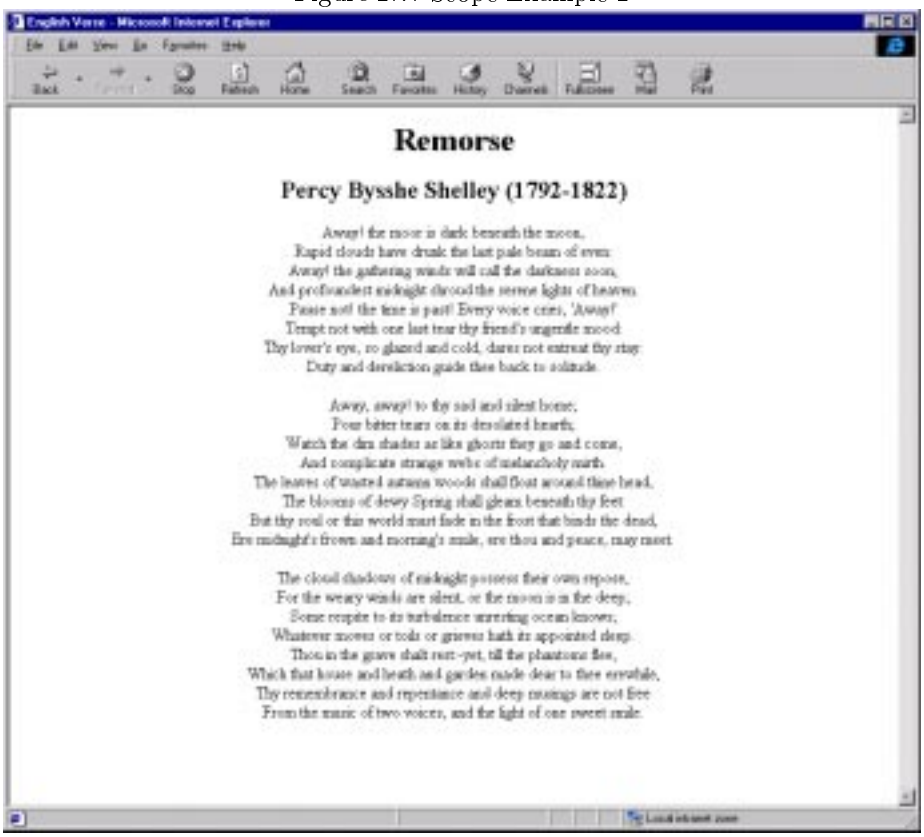
Figure 2.8 shows an example of several hypertext documents.

The three documents on the left all belong to a website containing information about English verse. The document on the right belongs to a different website about literature in general. However, it is possible to navigate from one website to the other through the use of hyperlinks. The term *different website* implies the following: different server name, hypertext authors, style sheets etc.

Now imagine the following: The user starts with the *introduction* page of the *English Verse* website. From there he navigates to *poem 1*. Let's assume that the poem is rendered with a small font that is hard to read for the user. Therefore he overrides the author's style sheet (style sheet A) that is referenced by the document and instructs the browser to render the document with the user's style sheet, which will render the poem with a bigger font. This instruction is not offered by today's major browsers, but it is described in the CSS specification [BLLJ98], for example.

Now the user navigates to *poem 2*. This document also contains a reference to an author's style sheet. There are two possibilities how the browser can behave in this situation: It can render the document with the user's style sheet, as the user explicitly instructed the browser to do so with the previous document, or it

Figure 2.7: Scope Example 2

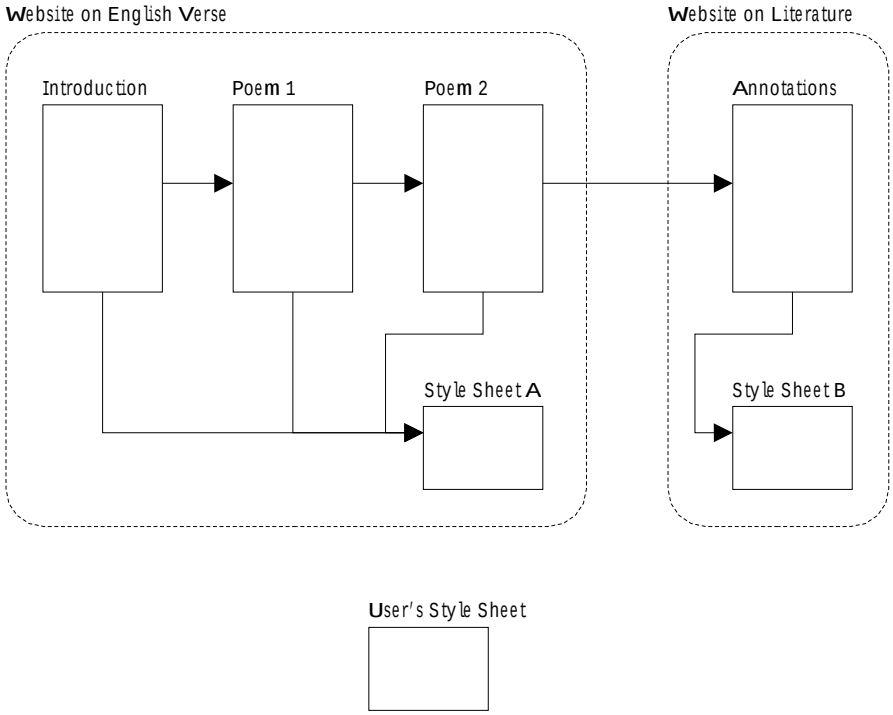


can render the document with the author's style sheet, as the document contains a reference to it. The desired behaviour is the first one, of course, because the user doesn't want to have to change the style sheet each time he reads another poem.

After having read the poem, the user navigates to the *annotations* page, which belongs to a different website. This document contains a reference to an author's style sheet (style sheet B) as well. Again, there are two possibilities how the browser can behave in this situation: It can render the document with the user's style sheet, as the user explicitly instructed the browser to do so before, or it can render the document with the author's style sheet, as the document contains a reference to it. In this case, the desired behaviour is the second one, of course, because the user doesn't need the bigger font anymore. It may even be impossible to apply the user's style sheet here, as the document from the general literature website may use a different namespace, element names etc. than documents from the English verse website.

It is impossible for the browser to automatically fulfil the user's desire in both cases, because it has no knowledge about the concept of *websites*. It cannot recognize when the user navigates "inside" the English verses and when he "leaves" them. The browser could infer the difference between the two websites from the server names, but this approach would create a dependency between

Figure 2.8: Scope Example 3



website membership and document storage. For example, the browser could define all documents from server `www.pms.informatik.uni-muenchen.de` to form one group of documents. Doing so, it would neither be possible to distribute a group of documents over more than one server, nor could two groups of documents be stored on the same server. Therefore a different method for defining groups of documents is needed.

2.4.2 Virtual Documents

The examples given above illustrate two problems of today's world wide web's hypertext models, namely HTML and XML. The first one is the problem of *context information*, that is hypertext documents are not able to specify any information about the relationship to other documents, such as described in the frame example. The second problem is a *scope* problem, that is hypertext documents are not able to specify information that is valid for other documents than themselves.

In a more abstract way, one could describe both problems as follows: With today's hypertext models it is not possible to define constructs at a level between a single *page* (that is a hypertext document) and the whole *world wide web*. To define a property of a single page, for example the font size, it is possible to define that property in the source document of that page. To define the font size of the whole world wide web, it is possible to change the browser's settings for that property. What is missing is the concept of a group of pages, or a

virtual document. With the help of a virtual document, it would be possible to change the font size of a set of pages, either by defining this property in the instantiation of a virtual document or by changing the browser's settings for that virtual document.

The concept of virtual documents is not only necessary to solve technical problems like the ones described so far, but it is also important for proper data modeling. It gives hypertext authors means to express the fact that a set of document forms *one entity* from several points of view. For example, consider a set of documents, each of which describing one page of an anthology of English verse. The whole set of documents forms a virtual document. This approach is analogous to a *book*, which is simply a set of single *pages*, but is considered to be one single entity as a whole.

2.4.3 Implementation

One problem is to define properties that apply to all members of the group or to the group as a whole. It is outside the scope of this text to define a general mechanism for defining virtual document properties, but two possible approaches will be stated: First, a generic markup language like XML could be used to model context information as described in the example. Alternatively, a general metadata mechanism like the Resource Description Framework (RDF) could be used.

The other problem is how to specify the *members* of a virtual document. As described above, it is important not to mix up the concepts of semantic relationships between documents and storage of the documents. Therefore, virtual documents cannot be defined in terms of server names or URIs. Instead, two other solutions are presented here.

Implicit Virtual Documents

This approach follows an idea that is already defined in the HTML standard. Using a `link` element in the head of the document with the `type` attribute set to `Start`, it is possible to define a hyperlink which "refers to the first document in a collection of documents". Doing like this, all pages belonging to the same virtual document could include a hyperlink to the same URI, which would act as an identifier for that virtual document. (This is analogous to URIs defining XML namespaces.) With such an identifier, the browser can decide whether a hypertext document belongs to a certain virtual document or not. The HTML approach can be converted to XML in exactly the same way as it has been done with the `xml-stylesheet` processing instruction at the beginning of this chapter:

A single hypertext document can be associated with a virtual document by using a processing instruction whose target is `dt-virtual-document` (`dt` for *diploma thesis*).

The following pseudo attributes are defined:

```
href CDATA #REQUIRED
```

```
type CDATA #REQUIRED
title CDATA #IMPLIED
media CDATA #IMPLIED
charset CDATA #IMPLIED
```

This processing instruction follows the behaviour of the `link` element described in the HTML specification.

The following example shows an XML document that is associated with a virtual document:

```
<?xml version="1.0" encoding="iso-8859-1"?>

<?dt-virtual-document href="../anthology.xml"?>

<page>
  <title>Percy Bysshe Shelley (1792-1822)</title>

  <!-- document content to follow -->
</page>
```

This example defines the XML document as a member of the virtual document `../anthology.xml`.

The approach using a processing instruction to define membership of a virtual document is implicit: There's no place where all members of the virtual document are listed together. This has the advantage that hypertext authors do not need special privileges to associate their documents with a virtual document, as it would be the case with an explicit virtual document definition (see below).

Explicit Virtual Documents

As an alternative to the implicit approach presented above, virtual documents could also be defined explicitly. In this case, there is a place where all members of the virtual document are listed together. As hyperlinks are used to create an association between several documents, they could also be used as the definition of a virtual document:

A hyperlink with a special arc role with the value of `dt-virtual-document` (`dt` for *diploma thesis*) defines a virtual document. The set of participating resources of the hyperlink is equal to the members of the virtual document.

The following example shows an XLink hyperlink which defines a virtual document:

```
<link xmlns:xlink="http://www.w3.org/1999/xlink/namespace/"
xlink:type="extended" xlink:role="dt-virtual-document">

  <locator xlink:type="locator"
  xlink:href="keats.xml"/>
```

```
  <locator xlink:type="locator"
  xlink:href="shelley.xml"/>

  <locator xlink:type="locator"
  xlink:href="tennyson.xml"/>

  <locator xlink:type="locator"
  xlink:href="wordsworth.xml"/>
</link>
```

This example defines a virtual document which consists of the hypertext documents `keats.xml`, `shelley.xml`, `tennyson.xml` and `wordsworth.xml`.

This explicit approach has the advantage that virtual document authors can protect their virtual documents from others, as other people do not have privileges to change the virtual document definition.

As both the explicit and the implicit approach have their advantages, a *hybrid* approach may be the best solution: Virtual documents have to be defined explicitly, but it is possible to allow hypertext documents to be added implicitly.

2.5 Source Document and Presentation Document

This section discusses a problem that arises when a transforming style sheet language like XSL is used together with a generic markup language like XML and a hyperlink language like XLink to render hypertext documents.

2.5.1 Example

To illustrate the problems discussed in this section, an extensive example is used. It consists of an XML source document, an XLink linkbase, an XSLT style sheet and an HTML presentation document, which results from the transformation of the source document with the style sheet.

The content of the XML source document is represented in *data* form:

```
<library>
  <author>
    <surname>Wordsworth</surname>
    <forename>William</forename>
    <born>1770</born>
    <died>1850</died>
  </author>

  <poem title="Daffodils" author="Wordsworth">
    <stanza>
      <line>I wander'd lonely as a cloud</line>
    </stanza>
```

```
  </poem>

  <poem title="The Solitary Reaper" author="Wordsworth">
    <stanza>
      <line>Behold her, single in the field,</line>
    </stanza>
  </poem>
</library>
```

This document models part of a library: Surname, forename, date of birth and death of an author are given, as well as the title of two of his poems together with their first lines.

The following XLink linkbase defines several hyperlinks that have one or more participating resources that reside in the source document from above:

```
<linkbase xmlns:xlink="http://www.w3.org/1999/xlink">
  <link xlink:type="extended">
    <locator xlink:type="locator" xlink:href="index.xml#
      xpointer(heading[1])"/>
    <locator xlink:type="locator" xlink:href="index.xml"/>
    <locator xlink:type="locator" xlink:href="author-index.xml"/>
    <locator xlink:type="locator" xlink:href="title-index.xml"/>
    <locator xlink:type="locator"
      xlink:href="first_line-index.xml"/>
  </link>

  <link xlink:type="extended">
    <locator xlink:type="locator" xlink:href="wordsworth.xml#
      xpointer(poem[1])"/>
    <locator xlink:type="locator" xlink:href="wordsworth.xml#
      xpointer(poem[1]//line)"/>
  </link>

  <link xlink:type="extended">
    <locator xlink:type="locator" xlink:href="wordsworth.xml#
      xpointer(author)"/>
    <locator xlink:type="locator" xlink:href="wordsworth.xml#
      xpointer(poem[1])"/>
    <locator xlink:type="locator" xlink:href="wordsworth.xml#
      xpointer(poem[2])"/>
  </link>
</linkbase>
```

The first hyperlink associates the first `heading` element in an XML source document with several other source documents. The next hyperlink associates the first poem (as a whole) with the first line of the poem. The last hyperlink defines an association between the author and the two poems.

The following XSLT style sheet defines a transformation from the XML source document to a presentation document represented in HTML:

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="library">
  <html>
    <head>
      <title/>
    </head>
    <body>
      <h1>
        <xsl:value-of select="author/forename"/>
        <xsl:value-of select="' '"/>
        <xsl:value-of select="author/surname"/>
        (<xsl:value-of select="author/born"/>
        <xsl:value-of select="'-'"/>
        <xsl:value-of select="author/died"/>)
      </h1>

      <h2>Index of Titles</h2>
      <xsl:for-each select="poem">
        <h3><xsl:value-of select="@title"/></h3>
      </xsl:for-each>

      <h2>Index of First Lines</h2>
      <xsl:for-each select="poem">
        <h3><xsl:value-of select="stanza/line"/></h3>
      </xsl:for-each>
    </body>
  </html>
</xsl:template>

</xsl:stylesheet>
```

This style sheet is a typical XSLT style sheet, as it containts the following three items:

- transformation of the source document's content from data form to document form
- style parameters
- layout parameters

If the source document is transformed using the style sheet, the result will be the following presentation document, represented in HTML:

```
<html>
  <head>
    <title/>
  </head>
```

```
  <body>
    <h1>William Wordsworth (1770-1850)</h1>

    <h2>Index of Titles</h2>

    <h3>Daffodils</h3>
    <h3>The Solitary Reaper</h3>

    <h2>Index of First Lines</h2>

    <h3>I wander'd lonely as a cloud</h3>
    <h3>Behold her, single in the field,</h3>
  </body>
</html>
```

Note that the presentation document does not contain any hyperlinks so far, it results from transforming the source document with the style sheet only.

The problem is as follows: Given the linkbase from above, how can the browser know that some of the list items of the presentation document should be participating resources of a hyperlink? And which elements exactly should be used as participating resources? One possibility is to use only the `forename` and `surname` elements, but the `born` and `died` elements or even all elements together could be used as well. This question will be discussed in more detail in the following.

2.5.2 Limitations of the XSLT Approach

The problem described above is specific to XSLT. In order to better understand this, a short summary of the concepts behind XSLT is given:

In today's XML world, hypertext documents consist of three different and independent concepts: content, style/layout and hyperlinks. In the former HTML 3.2 world, all of these three concepts were mixed together into a single language. Also, one document consisted of a single file containing data from all of these three concepts. Later, in HTML 4.0, style and layout data has been outsourced from the language and also from the document file. Style and layout data now has its own language (CSS) and is stored in its own file, the style sheet. Nowadays, in the new XML world, hyperlinks have also been outsourced into their own language (XLink) and this language also offers the possibility to store hyperlinks in a file distinct from the source document, called linkbase. Another change from HTML to XML is the style sheet language: CSS style rules leave the source document and therefore the document structure unchanged, whereas XSLT allows drastic restructuring transformations, such as resorting or inverting a table, leaving elements out and duplicate elements. The XSLT templates define a transformation from the XML source document into a target language (FO), which contains the data from the document as well as style and layout information. Because of the absence of tools for the FO language at the time of writing this text, HTML in conjunction with CSS is widely used as target language of the transformation.

A browser takes as input an XML source document, an XSLT stylesheet and an XLink linkbase. The transformations defined by the XSLT stylesheet are applied to the XML document in order to produce an FO document as output suitable for being rendered by the browser. Note that there are two different levels involved in this discussion: source level and presentation (rendering) level. The XML document is at the source level, whereas the FO (currently in general HTML) document resulting from the XSLT transformation is at the presentation level. But the latter one represents the transformation of the XML document only, hyperlinks aren't involved so far. Now three different questions arise:

- The first problem is that data not contained in the source document, but for example in the linkbase is also needed for rendering. How can these data, that are expressed in term of the document prior to the XSLT transformation be ported to the document resulting from the XSLT transformation?

- The second problem is that the user can interact with the presentation document, but an application needs to know the corresponding source level elements. How can interactions be performed on the XSLT transformed document when the necessary information is available in the document prior to the XSLT document?

  These first two problems are of an opposite nature: The first one has to do with transformation from source level to presentation level, the second one with transformation from presentation level to source level.

- The third question is about the level at which the hyperlinks should be specified: source level or presentation level?

In the following, we'll have a look at this last question first.

2.5.3 Source-Level and Presentation-Level Hyperlinks

There are two possibilities for incorporating the hyperlinks from the linkbase so as to implement them: Either on the source document or on the presentation document that results from the transformation of the source document using an XSLT stylesheet.

Incorporating hyperlinks from the linkbase directly into the presentation document would have an advantage with respect to processing: It would be easier to generate hyperlinks automatically in the presentation document, without the need of the style sheet and therefore the need for the style sheet author to generate the hyperlinks manually. This is because there is no need to transform the hyperlinks from source to presentation level any more, as they are already specified in terms of the presentation level from the very beginning. The major disadvantage of this approach, however, is of a semantic nature: typically, links are not understood as presentational features of an XML document, but as information about semantic relationships between elements. For this reason it does not make much sense for a document author to specify links in terms of the presentation level. Moreover, the might be several distinct presentation

documents for one source document. Expressing the hyperlinks in terms of the several presentation documents would lead to redundancy and be error prone.

Incorporating hyperlinks from the linkbase into the source document is in fact the only reasonable way to specify hyperlinks, as otherwise there would be a strong dependency between hyperlinks and style sheet, contradicting the data modelling goals of XLink. The definition of hyperlinks would heavily rely on the transformations that are carried out by the style sheet and not represent mere semantic relationships of elements any more.

So, from now on, we assume that hyperlinks are defined on elements of the source document only. Thus, hyperlinks have to be transformed, too, when the presentation document is generated as a transformation of a source document. In the following, transformation of hyperlinks will be covered in more detail.

2.5.4 Rendering

The browser renders elements that form the participating resource of a hyperlink in another manner than other elements: Hyperlink text will be underlined or rendered in a different color than normal text, for example. Or, an arrow ($\rightarrow$ like this) may be inserted in front of hyperlink text. Now, if the source document is translated into a presentation document, how can the browser know which text to underline or render in a different color? Typically, the XSLT transformation takes the source document as its only input, as described above. But data about hyperlinks might also be stored separately, in a linkbase. If all hyperlinks are external to the source document, the presentation document will not contain any hyperlinks, simply because the XSLT style sheet would not generate any. There are several possibilities to overcome this problem.

Semi-Automatic Transformation

The first approach is to give the responsibility of generating hyperlinks to the style sheet author. In this case, the XSLT style sheet has to ensure that the presentation document contains the desired hyperlinks. This gives the highest level of control about how hyperlinks are generated to the user or style sheet author, respectively. But usually, an XSLT style sheet has only one document as input (the XML source document), whereas this approach assumes that it has access to the linkbase as well. This technical restriction might be overcome in two ways.

Hyperlink Compilation of Source Document and Linkbase One possibility for the style sheet to access hyperlink information is to precompile the linkbase, that is, inbound or third-party hyperlinks, into the source document. As result, we get a document that contains outbound hyperlinks only. In this way, the style sheet could easily generate hyperlinks in the output document, as the only input would be the precompiled source document. However, the target language would have to support hyperlinks. XLink, for example, is specified using only attributes, and attributes can be added to arbitrary elements. But, if the participating resource of a hyperlink covers only part of the content of

an element, for example one word in a paragraph, a new element has to be inserted as "holder" of the attributes needed for an XLink hyperlink specification. This element would have as its content the word that makes up the hyperlink's participating resource. If the target language of the style sheet transformation does not support such elements, there is no way to express arbitrary XLink hyperlinks. It might also happen that insertion of new elements would invalidate existing hyperlinks. For example, if the participating resource of a hyperlink is defined as the third child of a certain element, then inserting an element before the third child will make the hyperlink point to a different than the desired location. As a consequence, it might be necessary to restrict XLink fragment identifiers to a subset of XPointer.

Direct Linkbase Access The other possibility is to access the linkbase directly during the XSLT transformation, using the XSLT `document` function. However, this approach comprises a new problem: XSLT offers no way to use an expression that defines the participating resource of a hyperlink as an expression for template matching. This problem has two different parts: First, the expression language used by XSLT differs from XPointer, which is used by XLink as language to describe fragment identifiers. However, the difference between those two languages is not very big, as both are extensions of XPath. Second, XSLT has no support for defining template matching expressions as a result of a former transformation. This could be solved if two transformations were carried out in sequence, however. The first transformation would generate the template matching expressions from the hyperlink data, then the second transformation would apply these templates to the source document. These templates, however, would match expressions that describe content that is neither contained in the source document nor in the linkbase, if the participating resource of a hyperlink covers only part of the content of an element (similar case to above). This concept of "virtual content" is not consistent with the XSLT design principles and is a result of the difference between the XSLT expression language and XPointer.

Compared with the compilation approach described above, that apporach seems to be much more natural and better fit into the other concepts of XML, XSLT and XLink.

Automatic Transformation

The second approach does all the hyperlink handling automatically. The browser will insert hyperlink elements into the presentation document by itself, considering all hyperlinks from the linkbase. Given our assumption from above, namely that hyperlinks are defined on elements from the source document, they will be invalid on the presentation document that results from the stylesheet transformation. The two document trees may containt elements from different namespaces and have a completely different structure. Elements may be inserted, deleted, merged, split, rearranged and so on. It is not clear whether XPointer fragment identifiers could be transformed automatically so that they point to the "intended" elements in the presentation document. Also, it has to be defined what is to be understood as "intended" first. Moreover, some

sort of control should be given to the user or style sheet author, for example what to do with hyperlinks that point to elements that are removed during the transformation process.

Even if this approach seems to be the most simple one for the style sheet author, as with this approach he would not have to cope with hyperlinks, it is not reasonable to retain it because it poses problems for which no reasonable solution seems possible. For example, if an element of the presentation document results from copying only part of a source-level element, which is the participating resource of a hyperlink, both copying the hyperlink as well as not copying it are reasonable behaviours of the browser depending on the semantics of the source-level element and of the hyperlink. Therefore it is indispensable that a transformation intended for style/layout also specifies how hyperlinks are ported from the source document to the presentation document, which is equal to the compilation approach from above.

2.5.5 Input Events

The user is able to click, select or otherwise manipulate elements from the document that is rendered by the browser. The problem is, that the object the user interacts with represent elements of the presentation level, whereas many applications need to know the "corresponding" elements of the source document. Again, the meaning of "corresponding" has to be specified first. This problem is somewhat the opposite of the rendering problem described above, as it is about a back transformation from presentation level to source level.

Semi-Automatic Transformation

This approach again gives the responsibility of transforming presentation level elements into source level elements to the style sheet author.

Source Level Identifiers The first possibility is to add an identifier to each element of the presentation document, which somehow refers to the corresponding source level element. Care has to be taken that these identifiers do not conflict with other elements in the document, as described above. So the best way would be to use attributes of a unique namespace for the identifiers. The identifiers could consist of XPointer expressions that depict the "corresponding" source elements of the element of the presentation document to which a certain attribute belongs. Again, it might be useful to allow only a subset of the XPointer language to be used. The identifiers would then be the definition of the meaning of "corresponding".

Reverse Transformation Alternatively, the stylesheet author could specify a set of transformation rules that work in the opposite direction of the XSLT style sheet transformation, namely from presentation level to source level. These rules could be specified as a set of XSLT templates and be stored in a different file. If an application needs a set of source elements that "correspond" to a set of elements from the presentation document selected by the user, the browser

could apply the reverse transformation sheet to the latter set and the result would be the first one. Here, the reverse templates would be the definition of "corresponding".

The reverse transformation approach seems to be the more natural one, as it represents simply the opposite of the style sheet transformation. For each "forward" transformation rule, the style sheet author has to specify a "reverse" transformation rule as well. Style sheet authoring tools could automatically suggest such a reverse transformation rule, which can then be altered by the style sheet author in order to meet his needs.

Automatic Transformation

Again, automatic transformation would be easy to use, but is by no means easy to conceive. The browser has to build its own reverse transformation rules from the "forward" transformation rules contained in the XSLT style sheet. It is not clear whether this could be done completely automatically, if a reverse transformation is always unique and if all possible XSLT transformations are allowed. Again, some sort of control should be given to the user or style sheet author to define the meaning of "corresponding".

As in the case of rendering, it is not reasonable to use the automatic transformation approach. Again, the term "corresponding" cannot be inferred automatically, so reverse transformation rules have to be specified by the user or the style sheet author.

Chapter 3

Advanced Browsing Functionalities

The previous chapter presented a low-level view of a hypertext model. In contrast to that, an application-driven view does not distinguish between terms like data, document, layout and others, which belong to *basic browsing*, but specifies its needs in terms like document view, reader's view, adaptivity, user preferences, browser modes and many more, that is, *advanced browsing* functionalities.

This chapter proposes and describes two such advanced browsing functionalities in detail, namely *browser modes* and *reader's views*.

3.1 Browser Modes

This section describes the concept of *browser modes*, which allows the user to switch between several sets of sheets. This concept is strongly based on the distinction between hyperlink model and browsing model, and on the notion of the hypertext model, which were all introduced in the previous chapter.

The first two subsections show how browser modes can cover the browser's behaviour related to hyperlinks. The next subsection extends this approach to a generic browser mode model. The last subsection deals with deficiencies of XSL and CSS, which are both meant to be used as style sheet languages for XML.

3.1.1 Hyperlink Browsing Behaviour

With the XML Linking Language (XLink), traversal of a hyperlink can have several effects: The ending-resource of the hyperlink traversal may occur in the same window as the hyperlink, or in a new window, or it may be inserted in place of the hyperlink (so-called *stretch text*). The intention is to be able to define certain browser modes that control the behaviour of hyperlinks in a document. For example, a mode for presentation on paper cannot use any

concept of windows, and there might be a mode for small screens like those of mobile phones that restricts the number of windows open at the same time. To achieve this, the traditional way of defining a browsing behaviour for a hyperlink within the hyperlink itself is not appropriate. Therefore it was proposed in the previous chapter to distinguish between hyperlink model and browser model. There, a hyperlink defines a *semantic* relationship between data items, whereas the browser model defines the *behaviour* of such hyperlinks.

Browser modes aim at influencing the treatment of hyperlinks. Consider, for example, a *hyperlink to alternative descriptions of the current information* (see 2.1.2). In a *standard mode* the browser treats a hyperlink of this kind in such a manner that the source text remains unchanged and the target text appears in a new window. In a *paper mode*, where the window concept does not make much sense, the hyperlink behaves like stretch text with appropriate delimiters for the beginning and end of the insertion. In a *small screen mode* the number of windows might be restricted to two, say, a main window and an auxiliary window, and the treatment of the hyperlink depends on whether it is actuated from the main window or from the auxiliary window and on whether the target text is already displayed in one of the windows.

Browser modes generalize a functionality available with today's major browsers: They allow the user to render the target of a hyperlink in the same window as the hyperlink, or the hyperlink can be rendered in a new window. It is not unreasonable to assume that, in the long run, browsers will not have built-in browser modes like this, but will interpret specifications of browser modes formulated in some declarative language in the same way as style sheets are formulated in some style sheet language. The kind of independency postulated here is similar to that ensured by generic markup languages by distinguishing between structure and style/layout.

3.1.2 Implementation: Sheets

The previous chapter defined a method for connecting browsing sheets with a source document. Amongst others, the pseudo-attributes `title` and `alternate` were defined. These pseudo-attributes stem from the `title` and `ref` attributes defined for the HTML `link` element.

HTML distinguishes between three different kinds of style sheets that can be associated with a source document. This approach can be used with browsing sheets as well:

- There may be a number of mutually exclusive *alternate* browsing sheets.
- The author may specify one of the alternate browsing sheets as the *preferred* browsing sheet, which is the default selection of the alternate browsing sheets.
- There may also be a number of *persistent* browsing sheets, which must be applied by the browser in addition to any alternate browsing sheet.

Note that even though this behaviour is part of the HTML specification, none of today's major browsers does implement this feature.

Browsing sheets may be specified as one of these three types by setting the pseudo-attributes as follows:

- To specify a *persistent* browsing sheet, set the value of the `alternate` pseudo-attribute to `no` (the default) and do not set the `title` attribute.
- To specify a *preferred* browsing sheet, set the value of the `alternate` pseudo-attribute to `no` (the default) and specify a mode name with the `title` attribute.
- To specify an *alternate* browsing sheet, set the value of the `alternate` pseudo-attribute to `yes` and specify a mode name with the `title` attribute.

How can this be used to implement browser modes? Take a look at the following example:

```
<?xml version="1.0" encoding="iso-8859-1"?>

<?dt-browsingsheet href="sheet1.css" type="text/css"?>
<?dt-browsingsheet href="sheet2a.css" title="alpha"
  type="text/css"?>
<?dt-browsingsheet href="sheet2b.css" title="beta"
  alternate="yes" type="text/css"?>
<?dt-browsingsheet href="sheet2c.css" title="gamma"
  alternate="yes" type="text/css"?>

<page>
  <!-- document content to follow -->
</page>
```

In this example, the browsing model defined in file `sheet1.css` is *persistent*, that means it will always be applied by the browser to the source document. Also, the user has the choice to choose between the modes *alpha*, *beta* and *gamma*, where mode *alpha* is the default. The three browsing models are defined in the files `sheet2a.css`, `sheet2b.css` and `sheet2c.css`, respectively. The title attribute is simply used as the name of the mode for which the browsing sheet should be applied.

The browsing models are responsible for defining different hyperlink behaviour as described at the beginning of this section. The user can then choose the actual hyperlink behaviour he wants to use by selecting one of the browser modes. The mechanism for defining the default browser mode can be further improved by using the `media` pseudo-attribute as well, which is also described in the HTML specification.

3.1.3 Generic Browser Mode Model

The approach just described can be extended to allow browser modes to cover not only browsing sheets, but arbitrary kinds of sheets. This procedure is analogous to the one described in the previous chapter, where the distinction between

hyperlink model and browsing model was extended to comprise models at arbitrary level.

As described in the previous chapter, the browser applies a set of properties to the document content before rendering it. These properties include hyperlinks, hyperlink behaviour, style, layout and others. The idea behind browser modes is that the user can choose between several sets of properties. Each of such sets is called a browser mode. Each mode defines its own set of properties concerning hyperlinks, hyperlink behaviour, style, layout etc.

Have a look at the following example:

```
<?xml version="1.0" encoding="iso-8859-1"?>

<?dt-browsingsheet href="browsing1.css" title="alpha"
  type="text/css"?>
<?dt-stylesheet href="style1.css" title="alpha" type="text/css"?>
<?dt-layoutsheet href="layout1.css" title="alpha"
  type="text/css"?>

<?dt-browsingsheet href="browsing2.css" title="beta"
  alternate="yes" type="text/css"?>
<?dt-stylesheet href="style2.css" title="beta" alternate="yes"
  type="text/css"?>
<?dt-layoutsheet href="layout2.css" title="beta" alternate="yes"
  type="text/css"?>

<?dt-browsingsheet href="browsing3.css" title="gamma"
  alternate="yes" type="text/css"?>
<?dt-stylesheet href="style3.css" title="gamma" alternate="yes"
  type="text/css"?>
<?dt-layoutsheet href="layout3.css" title="gamma" alternate="yes"
  type="text/css"?>

<page>
  <!-- document content to follow -->
</page>
```

This example defines three different modes. These modes do not define only different hyperlink behaviour, as in the example from the previous subsection, but also different style and layout of the source document: With mode *alpha*, the default mode, the sheets `browsing1.css`, `style1.css` and `layout1.css` are applied to the source document. The sheets for modes *beta* and *gamma* are `browsing2.css`, `style2.css`, `layout2.css` and `browsing3.css`, `style3.css` and `layout3.css`, respectively. This enables the document author not only to offer different hyperlink behaviour to the user, but also to accompany this hyperlink behaviour with different style (that is properties like fonts and colors) and layout. This enables the user to quickly distinguish the different modes.

The values of the `title` attribute serve as the definition of the browser modes. Due to browser modes, it is not possible to use an arbitrary combination of the

sheets that are referenced by the source document, but only those combinations defined by the browser modes.

Also, it would be possible to associate a different set of linkbases with the source document for each mode. Linkbases are usually associated with a source document using a hyperlink with a special arc role instead of a processing instruction (see section 2.2). This approach can also be extended to browser modes. It is only necessary to introduce in the XLink hyperlink specification attributes that play the role of the pseudo-attributes `title` and `alternate` of the processing instructions that associate sheets with source documents.

In the same way as with other sheets, the transformation of the content from data form to document form can vary from mode to mode. This feature allows that not only the *rendering* of the source document is controlled by browser modes, but also the document *content*. This means that the concept of browser modes can also be used to implement different *document views*.

Document views allow the user to choose between a given number of different representations of a document. For example, consider a document which represents an English poem. One document view may present the poem as it is, for the experienced reader. Another view may present the poem together with annotations for the novice reader. A third view may present the poem together with explanations of difficult words for foreign readers. Document views play an important role within the field of *Intelligent Tutoring Systems* [BEG99], which make use of several different user models.

3.1.4 Deficiencies of XSL and CSS

The intention of a browsing model is to control hyperlink behaviour, for example whether or not traversal of a hyperlink opens a new window. (See the behaviour attributes of XLink.) The intention is that hyperlink behaviour may depend on the window in which the hyperlink was rendered. For example, hyperlink behaviour may differ whether the hyperlink was traversed while showing in a "main" window or when it was rendered in an "auxiliary" window (see above). Similarly, other hypertext properties, for example fonts or colors, may also depend on the window in which they are rendered. Generally, all properties that are applied to the source document should be able to depend on the window in which the document is rendered.

The problem with both CSS and XSL, the style sheet languages that are meant to be used with XML, is, that neither of them define means to control windows or to infer the window in which the document is being rendered. In order to implement sophisticated browser modes as described in this section, it is necessary to have expressive style sheet languages that exceed the capabilities of CSS and XSL. Note that in this section, the term *window* is meant to comprise *frames* as well. Both CSS and XSL have no support for frames, too.

Even if FO is meant to be used as *rendering language* for XML, today HTML is widely used for this purpose, because of the large number of tools and browsers available for HTML. But the HTML hyperlink model is much more restrictive than XLink, the hyperlink model meant to be used for XML. It is possible to simulate certain features, like replacing one n-ary XLink hyperlink with a

number of binary HTML hyperlinks. But some features, for example stretchtext, that are crucial for sophisticated browser modes as described in this section cannot be simulated, at least not with reasonable effort. Therefore it is desirable to use an extension of the HTML hyperlink model, which covers all possible hyperlink behaviours that XLink provides.

Both the extensions to XSL and CSS, that will be described in the following, are not a complete solution to all the problems described here. They are only suitable to enable browsers to make rendering of source documents and hyperlink behaviour dependent on the window that they are rendered in. But what is really needed would be a complete window-aware style and layout language that offers much more possibilities to the sheet author than the ones described here.

Extension to XSL

As mentioned earlier, XSL consist of two parts: a language for transforming XML documents (XSLT) and an XML vocabulary for specifying formatting semantics (FO). First, we have a look at XSLT.

The basic concept behind XSLT are patterns and templates. The patterns are matched against the source document to decide whether or not a certain template is instantiated. The language for the patterns is an extension to XPath [CD99], which does not contain any constructs that deal with windows.

A possibility to implement window selectivity is to provide an *XSLT extension function* which returns the name of the window in which the current element is to be rendered:

```
<xsl:template match="dt:window-name()='main-window'& a">
  <xsl:copy>
    <xsl:attribute name="xlink:show">
      embed
    </xsl:attribute>
  </xsl:copy>
</xsl:template>

<xsl:template match="dt:window-name()='auxiliary-window'& a">
  <xsl:copy>
    <xsl:attribute name="xlink:show">
      new
    </xsl:attribute>
  </xsl:copy>
</xsl:template>
```

In this example, the two templates associate source document elements with name `a` with two different hyperlink behaviours: If the name of the window in which the hyperlink is rendered equals to `main-window`, the `a` element will be equipped with an `xlink:show` attribute whose value is set to `embed`, that is the hyperlink will behave like stretchtext. If the hyperlink is rendered in a window called `auxiliary-window`, the `xlink:show` attribute is set to `new`, which means

that traversal of the hyperlink opens a new window. This example assumes that the `a` element is actually a participating resource of a hyperlink.

Extension to CSS

In the previous chapter it was shown how the CSS mechanism of attaching property/value pairs to source document elements can be extended to cover other than style and layout issues as well. This was done by simply extending the existing property/value vocabulary of CSS.

The basic concepts of CSS are selectors and rules. Rules define a set of property/value pairs and selectors decide to which elements of the source document these property/value pairs are applied. The selector mechanism does not contain any constructs that deal with windows.

One possibility to make CSS selectors window-aware is to include a *window selector*, labelled by an asterisk:

```
*main-window a
{
  show: embed
}

*auxiliary-window a
{
  show: new
}
```

This example shows the same behaviour as the XSLT example from above: If an element with name `a` is rendered in a window called `main-window`, the hyperlink will behave like stretchtext, if it is rendered in a window called `auxiliary-window`, it will open a new window on traversal. Again, it is assumed that the `a` element is actually a participating resource of a hyperlink.

Note that it is necessary for these two approaches to work that windows can have names. HTML hyperlinks allow windows to be named through the use of the `target` attribute, but XLink hyperlinks do not. This means that XLink has to be extended in such a way that it provides this functionality as well.

Both approaches presented here allow sheets to be processed differently depending on the window the source document is rendered in. This can only be a first step towards a fully window-aware style and layout language, however. For example, there is still no way to control windows if they are not opened through traversal of a hyperlink. In the case of XSL, for example, a new formatting object `fo:window` might be introduced to control windows. With CSS, a new value `window` might be introduced for the `display` attribute, for example.

3.2 Reader's View

This section describes the concept of a *reader's view*. This is an advanced browsing functionality which allows the user to adapt source documents to his

own needs. It is a kind of generalization of today's major browsers' bookmark and history functionalities.

The first subsection gives a general overview about this concept. The second subsection describes features of a reader's view. Next, it is shown how a reader's view can be implemented using XLink. Finally, problems of the whole reader's view approach are discussed.

3.2.1 Overview

The objective of this project is to implement and test various advanced browser functionalities. One of these functionalities is a so-called reader's view. The general idea of a reader's view is to give the user an instrument to adapt a document to his own needs within certain limitations given by the author of the document or by the browser.

This is somewhat similar to an XML editor, but with one important difference: A reader's view leaves the original source document unchanged. For example, when the user decides to leave out a certain paragraph of text (see below), this paragraph won't be erased from the source document, but the reader's view stores the information not to display this paragraph. There are several different concepts behind the idea of a reader's view.

There's also a connection between the concept of a reader's view and the concept of *document views*. Document views allow the user to choose between a given number of different representations of a document. For example, consider a document which represents an English poem. One document view may present the poem as it is, for the experienced reader. Another view may present the poem together with annotations for the novice reader. A third view may present the poem together with explanations of difficult words for foreign readers. The difference between such document views and a reader's view is that the document view were created by the *author* of the document. The reader's view, however, is created by the *reader* of the document while he is browsing it.

3.2.2 Features

Selection and Rearrangement

The nodes of the document tree may be selected or rearranged. For example, a certain element can be left out or put in front of another element. There may be some limitations in this point. For example, it may be forbidden to violate parent-child relationships or to select parts of an element (or node in the document tree) rather than the element (or node) as a whole. There are two ways how elements can be selected or rearranged: Either explicitly by the user or automatically by recording the browsing actions of the user.

Explicit construction of the reader's view allows the user to *pick* certain nodes from the document and add it to the reader's view. The nodes of the reader's view can be put in any order. This functionality is similar to an electronic *notebook*.

With the first method, there must be a way for the user to select and rearrange document nodes explicitly. The reader's view is meant to contain nodes of the *source document*, not the presentation document.

The second method of how to select and rearrange way will be explained in the following.

Browsing History

This feature is a kind of generalization of today's major browsers' history function. Traditionally, the history function of a browser is organized like a stack. If a new page is visited, it is put on top of the stack. If the user uses the browser's *back* function, then the stack pointer is set to the page one before the top of the stack. If the user uses the back function again, the stack pointer is set to the page before that and so on. But if the user visits now another page (instead of using the *forward* function), the stack is popped up to the current location of the stack pointer (that is the page last visited) and the new page is put on top the stack. This means that if the user decides to go back one page again, then all the pages previously visited after the first visit to that page are no longer on the history stack.

In contrast to this approach, the reader's view stores information about visited pages in a tree, not in a stack. Therefore no information will be lost, regardless of how the user navigates through the pages. In addition to saving the tree structure of the pages visited so far, information about the order in which they were visited is also stored. This order can then be recalled later. This makes it possible, for example, to put chapter three in front of chapter two, if this seems to be a more appropriate order to the user. In this way, this feature offers the user another way to rearrange a set of pages, namely by visiting them in the desired order.

It is also possible for the user to exclude certain pages from the browsing history. Consider a user browsing through a set of pages, for example an anthology of English verse. He visits some pages in a certain order he wants to recall afterwards, as described above. But apart from that, the user also *explores* the whole set of pages by browsing to pages that he does not want to recall later. For example, he might take a look at a poem and then decide that he does not want to include it in the browsing history, because it was not a specific poem he looked for. The reader's view has an option to turn on or off the automatic browsing history recording or to edit the recorded history later. In this way, this feature offers the user a way to select certain pages which he wants to include in the browsing history.

Creating New Elements

As third feature of a reader's view, the user can hyperlink his own newly created elements to the source document. These elements may contain text, hyperlinks or anything else that might be part of a document. Again, the original source document won't be changed. Instead, the reader's view stores the new elements and merges them with the original document using hyperlinks.

In this way, a reader's view can be seen as the generalization of today's major browsers' bookmark function. Traditionally, the bookmark function allows the user to store a set of hyperlinks. In addition to that, the reader's view allows the user to store document elements, for example pieces of text, as well. These document elements can be hyperlinked to elements contained in the original source document. Moreover, these elements can contain their own hyperlinks. These hyperlinks may point to elements contained in the original document or to elements that are only contained in the reader's view. This means that the user can build up his own document containing new text and hyperlinks as well as text and hyperlinks from the original source document.

3.2.3 Implementation: XLink

How can a reader's view with features describes as above be implemented? The goal is not only to leave the source document unchanged, but also not to copy it. If, for example, a reader's view contains all of a document except a certain paragraph, changes to the original document (except to that paragraph) should be reflected instantly in the reader's view as well. This means that the reader's view doesn't contain any elements of the source document, but only hyperlinks to it.

A reader's view is simply an XML document. This means that it can be viewed with any browser, even if it is not reader's view-aware. The only difference to *normal* XML documents is the process of creation.

If the user selects a certain node from the original document, an outbound hyperlink with the selected node as its ending resource is created. This hyperlink is inserted into the reader's view document. If the user wants a certain node not to be included in the reader's view, there is no hyperlink created for that node.

The reader's view uses a special feature of the XLink language: The hyperlinks' behaviour attributes `show` and `actuate` are set to `embed` and `onLoad`, respectively. This has the effect of *including* the hyperlink's ending resource instead of the hyperlink itself when the reader's view document is loaded into the browser. Therefore the user will see the reader's view document as if its nodes were copied from the original document, even though they are only referenced by hyperlinks.

If the reader's view is created automatically by recording the user's browsing actions, this can be done similarly to the approach described above: Each page visited will be inserted as a new hyperlink's ending resource into the reader's view document. In this case, the user does not want all pages to be displayed at once, but one after the other, in the order they were visited. Therefore, the hyperlinks' behaviour attributes are set to `new` or `replace` and `onRequest`, respectively. Thus the reader's view document can be used as a kind of *guided tour* through the user's browsing history.

In the case of elements created by the user, the reader's view document does not contain just hyperlinks, as the elements' content has to be stored as well. The elements' content is copied to the reader's view document as it was entered by the user. If the document is rendered, these elements will be displayed as well.

3.2.4 Problems

The concept of a reader's view involves two problems, namely problems related to node selection and rendering. These problems have already been described before, but they will be repeated here in the special context of reader's views.

Node Selection

As stated above, the user is able select nodes of the document in order to add them to the reader's view or to rearrange them.

If the browser offers to the user the possibility to select nodes of the presentation document, they have to be transformed into nodes of the corresponding source document. This issue has already been described in section 2.5.

Alternatively, the user may be able to select node of the source document directly. For example, the browser might display the source document tree together with the presentation document.

In both cases, the problem is that the source document nodes do not necessarily contain the information the user wants to select. This is due to the difference between data form content and document content, as discussed in section 2.2. Take a look at the following source document:

```
<library>
  <author>
    <surname>Wordsworth</surname>
    <forename>William</forename>
    <born>1770</born>
    <died>1850</died>
  </author>

  <poem title="Daffodils" author="Wordsworth">
    <stanza>
      <line>I wander'd lonely as a cloud</line>
    </stanza>
  </poem>

  <poem title="The Solitary Reaper" author="Wordsworth">
    <stanza>
      <line>Behold her, single in the field,</line>
    </stanza>
  </poem>
</library>
```

Its content is represented in data form. The corresponding presentation document looks like this:

```
<html>
  <head>
    <title/>
```

```
  </head>
  <body>
    <h1>William Wordsworth (1770-1850)</h1>

    <h2>Index of Titles</h2>

    <h3>Daffodils</h3>
    <h3>The Solitary Reaper</h3>

    <h2>Index of First Lines</h2>

    <h3>I wander'd lonely as a cloud</h3>
    <h3>Behold her, single in the field,</h3>
  </body>
</html>
```

Lets assume that the user wants to add the *Index of Titles* list to the reader's view. If he selects nodes of the presentation document (the upper `h2` element and the two following `h3` elements), these nodes will be transformed to source level nodes, for example using a *reverse transformation*.

If the user selects nodes of the source documents directly, he will select the two `poem` elements or only their `title` attributes, for example.

In both cases, the source level nodes do not contain the text *Index of Titles*, as this text does not stem from the source document, but from a sheet which transforms the source document from data form to document form. There is no way for source level elements to contain nodes of this sheet.

The solution to this problem already lies in the hypertext model that is used in this project and has already been introduced in the previous chapter. As a reader's view is an XML document, it may also be associated with sheets, that is document sheets, style sheets, browsing sheets etc. If the reader's view document is associated with the same sheets as the document it was created from, its source level nodes will be transformed to the same presentation level nodes as the original document's nodes. But the association of sheets with a reader's view document involves other problems, that will be discussed in the following.

Rendering

As stated above, the reader's view document can also contain references to sheets, for example document sheet, style sheet, browsing sheet etc. It *must* contain these references in order to be rendered properly and to reproduce the presentation document is was created from.

The problem is, that the nodes of the reader's view document may stem from more than one *original* document. These original documents may have contained references to different sheets, for example two different style sheets. How can the reader's view be rendered properly, if these two sheets are used together? The style sheet languages CSS and XSL define the results of applying more than one sheet to a single source document, but the two sheets involved here were not

designed to be used together, which may lead to unpredictable and undesired results.

Moreover, the original documents may have used different document type definitions (DTD) or XSchema definitions. It is no problem to copy nodes from two document with different DTDs into a single document, but this document cannot be validated any longer. It is in general not possible to automatically generate a single DTD from the two distinct DTDs.

Nodes of an original document can be added as child nodes of an existing reader's view document's node. Consider a *list item* node which is added to the reader's view as child of a *heading* node. The original document's DTD allows only *list* nodes to have list item children, and the original document's sheets are designed with this assumption. The reader's view is no longer valid, and it cannot be rendered properly using the original document's sheets.

There seems to be no easy solution to allow nodes from arbitrary source documents be freely added at any position into a reader's view document. At the beginning of this section, the concept of a reader's view was described as *to give the user an instrument to adapt a document to his own needs within certain limitations given by the author of the document or by the browser.* Such a limitation might be, that only nodes from a single source document may be added to a single reader's view and only at positions so that the reader's view document is valid. Alternatively, if any nodes are allowed at all positions, the reader's view document might be rendered only using a *minimal* set of sheets, which applies default values for all rendering properties. In this way, the reader's view document may be rendered as a tree, for example.

Whereas the first solution ensures that the reader's view is rendered like the original document, the second solution gives the user more freedom in creating the reader's view. The browser might offer two different modes to the user, which allow him to switch between these two behaviours and selecting the one which best suits his needs.

Chapter 4

Implementation of a Browser Toolkit

Within this project, a browser toolkit for XML, called the *DT Browser*, has been conceived and implemented in Java. The toolkit is based on the principles discussed in the previous chapters. The objective is that the toolkit will be suitable to implement and test various advanced browser functionalities. In addition to the functionalities discussed and implemented within this project, the toolkit is meant to be used and further developed for new functionalities in various future projects. Because of this future application and extension, main criteria of the implementation are modularization and configurability.

The first section in this chapter gives a general overview of the *DT Browser* implementation. The second section describes general design principles of the implementation in Java. The following sections describe the design of each package of the implementation. In the last section, details of the implementation itself are described.

4.1 Outline of the *DT Browser* Implementation

In the previous chapters, various concepts about hypertext and browsing were introduced. For many problems, alternative solutions were given, some of which might even coexist. The *DT Browser* does not implement all of these solutions, however. This is partly due to the limited scope of this project and partly due to the features of the third party software that is used as part of the implemetation.

4.1.1 Model of Displaying and Basic Browsing

The *DT Browser* implements the distinction between hyperlink model and browsing model (see section 2.1). The linkbase is specified in P-XLink; the browsing sheet in XSLT. Browsing sheets in CSS are not implemented. The linkbase is associated with the XML source document using a hyperlink with

a special arc role; the browsing sheet is associated with the source document using a processing instruction.

The hypertext model of the *DT Browser* implementation consists of the following parts (see section 2.2): content in either data or document form, hyperlinks, style and browsing behaviour. Note that the term *style* here covers the transformation of the content from data to document form, if necessary, style and layout issues, like the traditional XSL or CSS approaches. The target language of the style sheet is HTML, which is used to render the document. The style sheet is associated with the source document using a processing instruction. The term *browsing behaviour* here only means hyperlink behaviour, as described above. The source document must be specified in XML, the style sheet in XSLT. Information about linkbase and browsing sheet has already been given above.

Linkbase processing (see section 2.3) in the *DT Browser* implementation is done as follows: The browser's internal linkbase is the union of the virtual document linkbases of all documents the browser has processed so far. It is not possible to preload linkbases into the browser or to delete certain hyperlinks from the browser's internal linkbase.

The concept of document grouping (see section 2.4) is not implemented in the *DT Browser*.

For rendering, hyperlinks are specified on the source level (see section 2.5). Source document and linkbase are then compiled together. Input events such as selection of document nodes on the presentation level are not implemented. Rather the *DT Browser* displays the document tree on the source level to enable the user to select document nodes.

4.1.2 Advanced Browsing Functionalities

The *DT Browser* implements browser modes (see section 3.1). The browser modes cover style sheet and browsing sheet, not the linkbase. The mode name is specified by the `title` attribute of the processing instruction that associated the source document with the sheet. Extensions to XSL and CSS are not implemented, standard HTML 3.2 is used for the presentation document.

The reader's view (see section 3.2) of the *DT Browser* implementation allows the user to select nodes from the source document and add it to the reader's view document. These nodes are selected from a presentation of the source document tree. Also, new nodes can be created and added to the reader's view document. It is not possible to record the browsing actions of the user in a browsing history. Due to the limited capabilities of the XLink processor used by the *DT Browser*, the reader's view document contains copies of the source document's nodes instead of only hyperlinks to them. For rendering, the reader's view document is associated with a browsing sheet and a style sheet, which can be chosen by the user.

4.2 General Design Principles

The browser toolkit is implemented in Java. This section describes design principles of the implementation that don't apply to a specific class or package, but

can be found across several classes or packages or apply to the implementation as a whole.

4.2.1 Interfaces

For a complete description of interfaces, see the lesson *Creating Interfaces* from *The Java Tutorial* [CW98].

An interface is a declaration of a set of methods without implementation. A class that implements an interface has to implement all of the methods defined in the interface, this means it has to agree to a certain behavior. First, we shall have a look at an example of two classes and an interface.

Example

This example is taken from the *DT Browser* implementation, with slight changes to the code to make the essentials clearer. It shows how the implementation uses interfaces and their implementations.

The `OpenDocumentFromFileAction` class represents one of the actions of the *DT Browser*. It asks the user for a file containing an XML source document and renders it in the browser window. To get the filename, it makes use of the user interface represented by an `UI` object. The `OpenDocumentFromFileAction` class is implemented like this:

```
public class OpenDocumentFromFileAction
{
    private UI ui;

    ...

    public void actionPerformed( ... )
    {
        String file;

        if((file=ui.getDocumentFile())!=null)
        {
            // display the document ...
        }
    }
}
```

If the return value is not `null`, that is, a filename was entered or otherwise selected by the user, then the `OpenDocumentFromFileAction` will proceed processing the file.

An object that is aimed to be a user interface must implement the `getDocumentFile` method so that `OpenDocumentFromFileAction` can call it to request the filename from the user. This requirement is enforced through the data type of the object being used as user interface, namely `UI`.

The `OpenDocumentFromFileAction` makes use of an object representing the user interface. The data type of this object is `UI`, which is declared to be an interface:

```
public interface UI
{
    public String getDocumentFile();

    // other UI methods ...
}
```

The `UI` interface defines the `getDocumentFile` method but does not provide a method body. Classes that implement this interface must provide this method body, that is, they must implement `getDocumentFile`.

Any object of type `UI` implements this interface and can therefore be used by `OpenDocumentFromFileAction`. This means that it provides method bodies for all methods declared in the interface definition. Therefore an object of type `UI` implements the `getDocumentFile` method, satisfying the requirement from the `OpenDocumentFromFileAction` class.

The following class is an example implementation of the `UI` interface. The `DefaultWindowUIImplementation` class implements a window user interface:

```
public class DefaultWindowUIImplementation implements UI
{
    private JFrame browserFrame;

    ...

    public String getDocumentFile() throws MalformedURLException
    {
        JFileChooser documentFileChooser=
           new JFileChooser("demos");

        if(documentFileChooser.showOpenDialog(browserFrame)==
           JFileChooser.APPROVE_OPTION)
            {
                return documentFileChooser.getSelectedFile().
                       toURL().toString();
            }
        else
            {
                return null;
            }
    }
}
```

This class implements the `getDocumentFile` method of the `UI` interface, so that it can be called by the `OpenDocumentFromFileAction` class.

Interfaces and Abstract Classes

Interfaces are not very different from abstract classes. Both define a set of methods without their bodies. Because of some special features of the Java language, the *DT Browser* implementation does not use abstract classes.

The following `UI` *class* corresponds to the `UI` *interface* that was shown above:

```
public abstract class UI
{
    public abstract String getDocumentFile();

    // other UI methods ...
}
```

If `UI` is declared like this, all objects that wish to be used by `OpenDocumentFromFileAction` must be instances of a subclass of `UI`. These objects may already have a superclass, however. For example, the `DefaultWindowUIImplementation` class might be subclass of some other class. As Java does not support multiple inheritance for classes, the `DefaultWindowUIImplementation` class cannot be subclass of two classes. Therefore the *DT Browser* implementation uses only interfaces, for which Java does allow multiple inheritance.

Implementation

As mentioned above, an interface defines a certain behavior. A class that implements an interface agrees to implement the behaviour defined by the interface. The class must provide method bodies for all methods declared in the interface.

As an example, we show a possible alternative implementation of the `UI` interface. This time, the user interface is a textual user interface, that is, it will work on any terminal that is not able to display graphics:

```
public class TextualUIImplementation implements UI
{
    ...

    public String getDocumentFile()
    {
        System.out.print("Please enter the URI of an XML source
                          document: ");

        BufferedReader in=new BufferedReader(new
                          InputStreamReader(System.in));

        return in.readLine();
    }
}
```

The code from `OpenDocumentFromFileAction` remains unchanged. That is, if you want to use a textual user interface instead of the window user interface, all you have to do is to implement the `UI` interface in the appropriate way and instantiate the desired class (see the next section). There is no need to change any code that is calling `UI` methods.

Interfaces in the *DT Browser* Implementation

The *DT Browser* is a toolkit meant to be used and further developed for new functionalities in various future projects and therefore makes heavy use of interfaces which contributes to modularization and configurability. The `UI` interface presented in the example above is not the only interface used by the *DT Browser* implementation. In fact, there is quite a number of them, which can be seen in table 4.1.

As you can see, most interfaces have an implementation called `DefaultXXXImplementation`. Note that nowhere in the browser a reference to a specific implementation of an interface is used. In the example above, `OpenDocumentFromFileAction` only refers to an object of type `UI` and not of type `DefaultWindowUIImplementation`. This means that the browser is completely independent from the actual implementation of this interface.

Having the differences between an interface and its implementation in mind, one can say that the description of the interface API is a specification, whereas the description of the implementation API is a documentation.

4.2.2 Factories

For a complete description of factories, see the *Factory Method* section of *Design Patterns* [GHJV95].

Overview

Complex software systems use interfaces to define and ensure a certain behaviour of objects, as described above. A software system is responsible for creating these objects as well.

As an example, we look at a browser that uses a separate user interface to render source documents. The user interface is defined as interface `UI` as in the previous subsection. To create a browser with a window user interface, we define the class `DefaultWindowUIImplementation`, which implements the `UI` interface. The browser class is responsible for managing the user interface and will create an `UI` object upon creation of a new browser instance.

The browser class does not know which `UI` implementation to instantiate, because the actual class implementing the `UI` interface is application-specific. The browser class only knows *when* to create a new user interface, not *what kind* of user interface. The problem is that the browser class only knows about the `UI` interface, but interfaces cannot be instantiated.

A solution to this problem is the *factory* design pattern. It moves the knowledge of what kind of `UI` implementation to create out of the browser into a *factory*

Table 4.1: Interfaces in the *DT Browser* Implementation

Interface	**Implementation**	**Description**
`Browser`	`DefaultBrowserImplementation`	A `Browser` applies sheets to an XML source document and displays it to the user.
`Document`	`DefaultDocumentImplementation`	A `Document` is the abstraction of an XML source document.
`ReadersView`	`DefaultReadersViewImplementation`	A `ReadersView` is an XML document that consists of nodes copied from another XML source document and newly created nodes.
`SheetEngine`	`DefaultSheetEngineImplementation`	A `SheetEngine` transforms XML source documents to presentation documents according to a set of sheets.
`ActionFactoryListener`	*used as superinterface of `UI`*	An `ActionFactoryListener`, superinterface of `UI`, is used to register browser actions.
`BrowserAction`	`XXXAction`	All actions exposed by the browser must be an implementation of `BrowserAction`.
`BrowserLinkListener`	`DefaultBrowserLinkListenerImlementation`	A `BrowserLinkListener` is responsible for handling hyperlink events.
`CloseAction`	`CloseBrowserAction`	One action of the DT Browser has to be an implementation of the `CloseAction` interface, which is a subinterface of `BrowserAction`.
`MessageHandler`	`DefaultMessageHandlerImplementation`	A `MessageHandler` is capable of displaying messages such as errors and diagnostic messages to the user.
`LoadingUI`	`DefaultWindowLoadingUIImplementation`	A `LoadingUI` object is able to display information to the user while the browser is still loading.
`UI`	`DefaultWindowUIImplementation`	A `UI` object represents the browser's user interface.

class. This class is called *factory* because is it responsible for *manufacturing* objects. The factory class defines a `getUI` method which returns an object that implements the `UI` interface. This enables the browser class to instantiate an application-specific user interface without knowing its actual class.

The following are the participants of the factory design pattern:

- *Product* (`UI`) defines the interface of objects the factory class creates.
- *ConcreteProduct* (`DefaultWindowUIImplementation`) is an implementation of the Product interface.
- *Creator/ConcreteCreator* (`UIFactory`) defines the factory method (`getUI`), which returns an object of type Product. The factory method is called by an application (the browser) to create a Product object.

With the help of a factory class, it is possible for the browser class to create a user interface without the need to use application-specific code. It only deals with the Product interface and the Creator class and can therefore work with any ConcreteProduct class. Factory classes allow the browser to be used with different user interfaces without changing the browser's code.

Factories in the *DT Browser* Implementation

So far, it has not been described how a factory can create multiple kinds of products. With the approach of *parameterized* object creation the factory method takes a parameter that identifies the kind of object to create. In the example from above, the browser might support different kinds of user interfaces, for example a window user interface and a textual user interface. To achieve this, a parameter is passed to the factory's `getUI` method to specify the kind of user interface to create. This parameter is simply the name of the class implementing the `UI` interface that will be instantiated.

The *DT Browser* factory classes use a special mechanism to decide which class to instantiate:

- First, the system is checked for a system property which contains the name of the class to be instantiated. If it is set, its value is taken as class name.
- Otherwise, an instance of a default implementation is created.

For example, if you want to use the class `TextualUIImplementation` from the previous subsection to be used as implementation of the `UI` interface, you can invoke the *DT Browser* from the command line like this:

```
java -classpath dt-browser.jar \
     -Dde.uni_muenchen.informatik.pms.dt_browser.ui=
       TextualUIImplementation \
     de.uni_muenchen.informatik.pms.dt_browser.CommandLine
```

Table 4.2: Factories in the *DT Browser* Implementation

Factory	**System Property**
	Default Implementation
BrowserFactory	de.uni_muenchen.informatik.pms.dt_browser.browser
	de.uni_muenchen.informatik.pms.dt_browser.DefaultBrowserImplementation
DocumentFactory	de.uni_muenchen.informatik.pms.dt_browser.document
	de.uni_muenchen.informatik.pms.dt_browser.document.DefaultDocumentImplementation
SheetFactory	de.uni_muenchen.informatik.pms.dt_browser.sheetengine
	de.uni_muenchen.informatik.pms.dt_browser.sheet.DefaultSheetEngineImplementation
ReadersViewFactory	de.uni_muenchen.informatik.pms.dt_browser.readersview
	de.uni_muenchen.informatik.pms.dt_browser.readersview.DefaultReadersViewImplementation
MessageFactory	de.uni_muenchen.informatik.pms.dt_browser.messagehandler
	de.uni_muenchen.informatik.pms.dt_browser.message.DefaultMessageHandlerImplementation
UIFactory	de.uni_muenchen.informatik.pms.dt_browser.ui
	de.uni_muenchen.informatik.pms.dt_browser.ui.DefaultWindowUIImplementation
	de.uni_muenchen.informatik.pms.dt_browser.loadingui
	de.uni_muenchen.informatik.pms.dt_browser.ui.DefaultWindowLoadingUIImplementation

Table 4.2 shows all factory classes and the corresponding system properties used in the *DT Browser* implementation.

With the approach described here, it is possible to change the implementation of the `UI` interface without having to recompile the browser. The only class that has to be compiled is the user interface class `TextualUIImplementation`.

In future, the *DT Browser* might also include features to change the implementation of an interface while it is running. That is, the user is able to change the `UI` implementation and therefore the user interface while he is using the browser.

Sample Code

Each package from the *DT Browser* implementation has its own factory class which is responsible for creating objects from that package (see below). For example, the `UIFactory` class from the `de.uni_muenchen.informatik.pms.dt_browser.ui` package looks like this:

```
public class UIFactory
{
    public static final String UI_PROPERTY=
    "de.uni_muenchen.informatik.pms.dt_browser.ui";

    public static final String DEFAULT_UI="de.uni_muenchen.
     informatik.pms.dt_browser.ui.DefaultWindowUIImplementation";

    public static UI getUI()
    {
        String className;

        if((className=System.getProperty(UI_PROPERTY))==null)
        {
                className=DEFAULT_UI;
        }

        return (UI)Class.forName(className).newInstance();
    }

    ...
}
```

As each browser is associated with an implementation of `UI`, the following code can be found within the browser implementation:

```
public class DefaultBrowserImplementation implements Browser
{
    private UI ui;

    ...
```

```
    public DefaultBrowserImplementation()
    {
        ...

        ui=UIFactory.getUI();
    }
}
```

To be able to change the implementation of an interface while the browser is running, the `getUI` method from `UIFactory` might be changed as follows:

```
public static UI getUI(String className)
{
    if(className==null)
    {
        if((className=System.getProperty(UI_PROPERTY))==null)
        {
            className=DEFAULT_UI;
        }
    }

    return (UI)Class.forName(className).newInstance();
}
```

Further changes have to take place within the browser implementation to completely implement this feature.

4.2.3 Packages

For a complete description of packages, see the lesson *Creating and Using Packages* from *The Java Tutorial* [CW98].

Packages allow the programmer to bundle classes into groups. A package is simply a set of classes and interfaces. It provides for access protection and namespace management.

The classes and interfaces that are part of the *DT Browser* implementation are members of various packages that bundle classes by functionality. For example, document classes reside in the
`de.uni_muenchen.informatik.pms.dt_browser.document` package and
reader's view classes reside in the
`de.uni_muenchen.informatik.pms.dt_browser.readersview` package.

Package Names

The Java Tutorial gives the following information about package names:

> By Convention: Companies use their reversed Internet domain name in their package names, like this: `com.company.package`. Name

collisions that occur within a single company need to be handled by convention within that company, perhaps by including the region or project name after the company name, for example, `com.company.region.package`.

The package names in the *DT Browser* implementation follow the naming conventions of Java, as can be seen in the name of the `de.uni_muenchen.informatik.pms.dt_browser` package, for example. The hyphen of `uni-muenchen` has been transformed into an underscore. The naming conventions say nothing about hyphens and other characters allowed in a URI, but not inside a package name, though.

Packages in the *DT Browser* Implementation

All packages have a similar internal structure due to the interface and factory design principles. Each package contains one or more interfaces representing the concepts of that package, one or more implementations of each interface and one factory class. In addition to that, a package can contain classes which act as helper classes for the interface implementations. Table 4.3 gives an overview of all packages of the *DT Browser* implementation.

4.2.4 Exception Handling

As all Java applications, the *DT Browser* implementation needs to handle exceptions. Java requires that methods catch or specify all exceptions that can be thrown within that method. The *DT Browser* implementation has a special policy about when to catch and when to specify an exception.

Catch

A method can catch an exception by providing an exception handler for that type of exception. There is only a limited number of points in the *DT Browser* implementation where exceptions are caught:

Classes Invoking the Browser Classes belonging to this group may catch exceptions that occur while the browser is instantiated. Currently, the only existing class that invokes the browser is `CommandLine`. Technically, it has to catch all exceptions because the `main` method may not specify exceptions. If the browser is invoked from within an application, the class calling the `BrowserFactory.getBrowser` method is free to catch or specify any occurring exception. In the latter case, the exception has to be caught somewhere else within the application. Logically, the user has to be informed about any exception that occurs while instantiating the browser.

Browser Actions These are classes that implement one of the `BrowserAction`, `CloseAction` or `BrowserLinkListener` interfaces. Technically, they have to catch all exceptions as the `actionPerformed` (for

Table 4.3: Packages in the *DT Browser* Implementation

Package			
Description			
Interfaces	**Implementations**	**Factory**	**Helper Classes**
de.uni_muenchen.informatik.pms.dt_browser			
Contains classes and interfaces that represent the concept of a browser as well as means to invoke a browser instance from within various contexts.			
Browser	DefaultBrowserImplementation	BrowserFactory	CommandLine Utilities
de.uni_muenchen.informatik.pms.dt_browser.document			
Contains classes and interfaces that represent the concept of an XML source document.			
Document	DefaultDocumentImplementation	DocumentFactory	
de.uni_muenchen.informatik.pms.dt_browser.readersview			
Contains classes and interfaces that represent the concept of a reader's view.			
ReadersView	DefaultReadersViewImplementation	ReadersViewFactory	NodeData
de.uni_muenchen.informatik.pms.dt_browser.sheet			
Contains classes and interfaces that provide the browser with means of applying sheets to XML source documents as well as handling browser modes and document scopes.			
SheetEngine	DefaultSheetEngineImplementation	SheetFactory	ArtificialServletEnvironment
de.uni_muenchen.informatik.pms.dt_browser.action			
Contains classes and interfaces that represent the concept of actions as well as all actions of the current implementation.			
BrowserAction CloseAction BrowserLinkListener ActionFactoryListener	XXXAction DefaultBrowserLinkListenerImplementation	ActionFactory	SAXHandler
de.uni_muenchen.informatik.pms.dt_browser.message			
Contains classes and interfaces that provide the browser with means of displaying messages to the user.			
MessageHandler	DefaultMessageHandlerImplementation	MessageFactory	
de.uni_muenchen.informatik.pms.dt_browser.ui			
Contains classes and interfaces that represent the concept of a user interface, which is responsible for displaying the browser to the user.			
LoadingUI UI	DefaultWindowLoadingUIImplementation DefaultWindowUIImplementation	UIFactory	ClosingWindowListener

`BrowserAction` and `CloseAction`) and `hyperlinkUpdate` methods (for `BrowserLinkListener`) may not specify exceptions. For logical reasons, exceptions should be caught here because browser actions are the only connection between the application logic and the user interface. If the application logic classes did catch some exception, it wouldn't have any means to inform the user about that. Otherwise, the user interface classes shouldn't contain any active program parts. Therfore, all exception handling is done within the browser actions.

Helper Classes Classes belonging to this group may technically be forced to catch some exceptions, as the methods in which exceptions can occur may not be able to specify certain exceptions. As stated above, all exception handling is meant to be centralized within the browser action classes. There is no reason except the technical one why some exception should be caught in the helper classes.

Specify

If a method chooses not to catch an exception, the method must specify that it can throw that exception. All places except the ones stated above don't catch any exception, but specify them. That is, interfaces (because of technical reasons), their implementations (except browser actions) and factory classes never catch any exception. The technical and logical reasons for this behaviour have already been explained above.

Exceptions and Message Handlers

The `message(java.lang.Exception)` method from the `MessageHandler` interface is meant to inform the user about the occurrence of an exception. Each browser is associated with one `MessageHandler` object that is used for this purpose. The `CommandLine` class creates its own `MessageHandler` object to display the exception to the user, as the browser's message handler might not yet have been instantiated. All other classes use the browser's message handler, which is accessible through its `getMessageHanlder` method.

Currently, exceptions are just passed through to the message handler, which looks like this:

```
try
{
    ...
}
catch(Exception exception)
{
    messageHandler.message(exception);
}
```

Because the current `DefaultMessageHandlerImplementation` simply calls the exception's `printStackTrace` method, the occurrence of an exception leads to an output like the following:

```
org.xml.sax.SAXParseException: File "file:/mnt/users9/home/k/
  krausm/diplomarbeit/software/dt-browser/demos/abc" not found.
        at java.lang.Throwable.<init>(Throwable.java:96)
        at java.lang.Exception.<init>(Exception.java:44)
        at org.xml.sax.SAXException.<init>(SAXException.java:45)
        at org.xml.sax.SAXParseException.<init>
           (SAXParseException.java:56)
        at org.apache.xerces.framework.XMLParser.reportError
           (XMLParser.java:975)
        at org.apache.xerces.readers.DefaultEntityHandler.
           startReadingFromDocument
           (DefaultEntityHandler.java:512)
        at org.apache.xerces.framework.XMLParser.parseSomeSetup
           (XMLParser.java:303)
        at org.apache.xerces.framework.XMLParser.parse
           (XMLParser.java:860)
        at org.apache.xerces.framework.XMLParser.parse
           (XMLParser.java:900)
        at de.uni_muenchen.informatik.pms.dt_browser.document.
           DefaultDocumentImplementation.setDocument
           (DefaultDocumentImplementation.java:44)
        at de.uni_muenchen.informatik.pms.dt_browser.action.
           OpenDocumentFromFileAction.actionPerformed
           (OpenDocumentFromFileAction.java:44)
        at javax.swing.AbstractButton.fireActionPerformed
           (AbstractButton.java:1450)
        at javax.swing.AbstractButton$ForwardActionEvents.
           actionPerformed(AbstractButton.java:1504)
        at javax.swing.DefaultButtonModel.fireActionPerformed
           (DefaultButtonModel.java:384)
        at javax.swing.DefaultButtonModel.setPressed
           (DefaultButtonModel.java:256)
        at javax.swing.AbstractButton.doClick
           (AbstractButton.java:279)
        at javax.swing.plaf.basic.BasicMenuItemUI$
           MouseInputHandler.mouseReleased
           (BasicMenuItemUI.java:931)
        at java.awt.Component.processMouseEvent
           (Component.java:3733)
        at java.awt.Component.processEvent(Component.java:3562)
        at java.awt.Container.processEvent(Container.java:1173)
        at java.awt.Component.dispatchEventImpl
           (Component.java:2611)
        at java.awt.Container.dispatchEventImpl
           (Container.java:1222)
        at java.awt.Component.dispatchEvent(Component.java:2515)
        at java.awt.LightweightDispatcher.retargetMouseEvent
           (Container.java:2465)
        at java.awt.LightweightDispatcher.processMouseEvent
           (Container.java:2230)
```

```
        at java.awt.LightweightDispatcher.dispatchEvent
           (Container.java:2139)
        at java.awt.Container.dispatchEventImpl
           (Container.java:1209)
        at java.awt.Window.dispatchEventImpl(Window.java:923)
        at java.awt.Component.dispatchEvent(Component.java:2515)
        at java.awt.EventQueue.dispatchEvent(EventQueue.java:401)
        at java.awt.EventDispatchThread.pumpOneEvent
           (EventDispatchThread.java:109)
        at java.awt.EventDispatchThread.pumpEvents
           (EventDispatchThread.java:99)
        at java.awt.EventDispatchThread.run
           (EventDispatchThread.java:90)
```

In future, those exceptions may be converted to more human-readable messages like "XML source document file not found". This can be done in two ways: First, the exception may be converted within the application logic or user interface to other exceptions, like this:

```
try
{
    ...
}
catch(SAXException exception)
{
    throw new FileNotFoundException(exception.getMessage());
}
```

Then, the corresponding browser action would have to catch a `FileNotFoundException`. Second, the exception may be converted within the `Browser` interface into a string message:

```
try
{
    ...
}
catch(SAXException exception)
{
     messageHandler.message("XML source file not found");
}
```

If applicable, the application logic or user interface classes might also try to resolve certain exceptions through fallback mechanisms. For example, if a file cannot be opened, the user interface might ask the user again for a filename, as long as a valid filename is entered or the action is cancelled. For this reason, application logic or user interface classes may also catch exceptions, but the general reasons for when to catch and when to specify exceptions given above still apply.

4.3 Specific Design

This section describes the design of each package of the implementation. Classes that do not provide any features exceeding the ones described in the previous section are not mentioned.

4.3.1 Browser

The `de.uni_muenchen.informatik.pms.dt_browser` package contains classes and interfaces that represent the concept of a browser as well as means to invoke a browser instance from within various contexts.

Currently, there are only two ways to invoke a browser. First, from the command line, using the `CommandLine` class and second, from within another application, using the `BrowserFactory` class. In future, there should be more contexts from within which the browser could be invoked, for example as an applet or a bean.

The `Browser` Interface

A `Browser` is something that displays an XML source document to the user. This interface defines the methods any browser must implement. A browser consists of various parts: an XML document to be displayed, a sheet engine that transforms the source document into a presentation document, a reader's view, a user interface and a user interface for error messages. For the user, if for example a window user interface for controlling the browser is used, a browser is one or more windows which display an XML document, a reader's view and error messages.

This interface acts only as a collection of the various classes and interfaces from the application logic and user interface packages. Therefore only a set of `getXXX` methods is provided, which return an instance of the corresponding interface. The only exception to this is the `close` method, which notifies the `BrowserFactory` when this browser has been closed.

Currently, the reader's view is an integral part of a browser. This was done for two reasons. First, it is easier to manage the user interface if there's always an XML document and a reader's view present. Second, there's no ambiguity if a node from an XML document is copied to a reader's view about when there are more than one document and more than one reader's view present at the same time.

In future, the reader's view should simply become a subclass of `Browser`. Eventually other interfaces have to be subclassed as well, for example `Document`, to do justice to the requirements of a reader's view. In this case, the problem of the `UI` interface getting too complex would vanish, because the user interface of the XML document and the reader's view user became two separate instances of the `UI` interface. The problem of the ambiguity of source and target of an *add selected node to reader's view* operation would still have to be solved, yet.

The `Utilities` class contains several utility methods and constants used by the DT browser implementation. The methods and contants defined in this class are all `static`, that is this class is never instantiated. All methods and constants

that do not belong to any specific concept like document, sheet or user interface are placed in this class.

The `DefaultBrowserImplementation` class simply creates instances of the `Document`, `MessageHandler`, `ReadersView`, `SheetEngine` and `UI` interfaces using the respective factories and returns references to them by request to the various `getXXX` methods.

The `BrowserFactory` Class

The `BrowserFactory` class is a factory class for vending objects from the `de.uni_muenchen.informatik.pms.dt_browser` package. Currently, the only class vended by this factory is `Browser`.

This factory not only vends `Browser` objects, it also keeps track of the number of browsers open. Each time a new browser is vended, a counter is increased automatically. To decrease the counter, this factory has to be informed manually about the closure of a browser using the `closeBrowser` method.

The `CommandLine` Class

This class is meant to be used to create an instance of a *DT Browser* from the command line. This class is also used by the *start* tool 4.4.4 to invoke a browser. For example:

```
java -classpath dt-browser.jar: ... \
     de.uni_muenchen.informatik.pms.dt_browser.CommandLine
```

While loading and initializing a `Browser` instance, an instance of `LoadingUI` is displayed to the user. Any exceptions occuring during this process are handled by a `MessageHandler` instance.

Note that this class does *not* process any arguments passed in on the command line, but this may be added in future. For example, the URI of an XML source document to be displayed could be given as an argument.

4.3.2 Documents

The `de.uni_muenchen.informatik.pms.dt_browser.document` package contains classes and interfaces that represent the concept of an XML source document.

The `Document` Interface

A `Document` is the abstraction of an XML source document. This interface defines the methods any document must implement. Each browser is associated with exactly one document, which is the abstraction of the XML source document that is being displayed.

The URI of the XML source document may be set using the `setDocument` method. A subsequent call to `getDocument` will then return the XML source document parsed into a DOM tree.

The `DefaultDocumentImplementation` class simply creates and stores a DOM tree representation of an XML source document.

4.3.3 Reader's View

The `de.uni_muenchen.informatik.pms.dt_browser.readersview` package is the home for classes and interfaces that represent the concept of a reader's view.

The `ReadersView` Interface

A `ReadersView` is an XML document that consists of nodes copied from an XML source document and newly created nodes. This interface defines the methods any reader's view must implement. Each browser is associated with exactly one reader's view, which can hold nodes from the various source documents the browser displays over the time.

Currently, the reader's view provides functions for copying nodes from an XML source document to the reader's view or to create new text nodes and add them to the reader's view. Additionally, the reader's view can be loaded from and saved to a URI.

Currently, the reader's view is an integral part of a browser. This may change in future and the reader's view may become a subclass of `Browser`.

A `NodeData` object is the link between the presentational representation of a document tree of and the document tree itself. It simply holds the string representation of the node as well as a reference to the node of the document tree itself.

The user interacts with the reader's view document by interacting with a representation of the document tree, both of the reader's view document and the source document. `NodeData` objects are used as *user objects* of the tree nodes of the presentational representaion of the document tree, both of the reader's view document as well as the source document.

In future, the `NodeData` class may move to the `de.uni_muenchen.informatik.pms.dt_browser.ui` package as it is only necessary because of the special features of the user interface.

4.3.4 Sheets

The `de.uni_muenchen.informatik.pms.dt_browser.sheet` package contains classes and interfaces that provide the browser with means of applying sheets to XML source documents as well as handling browser modes.

To the browser, all documents that are applied to an XML source document, are called sheets. Currently, this comprises browsing sheets, style sheets as well as linkbases. In future, there may be added more types of sheets due to a more fine-grained hyperlink model.

The `SheetEngine` Interface

A `SheetEngine` transforms XML source documents to presentation documents. This interface defines the methods any sheet engine must implement. Each browser is associated with exactly one sheet engine, which transforms an XML source document into the presentation document that is displayed.

This version of `SheetEngine` processes three sheets: linkbase, browsing sheet and style sheet. The linkbase is an XLink external linkbase. The XML source document can contain references to one or more linkbases using an XLink hyperlink which type equals `external-linkset`. Both browsing sheet and style sheet are XSLT sheets. An XML source document may contain references to these sheets using the `dt-browsingsheet` and `xml-stylesheet` processing instructions respectively.

When a new source document is loaded into the browser, the `parseDocument` method examines it for any references to sheets, and stores this information in the sheet engine. A subsequent call to `transform` will then transform the source document into a presentation document. The `transform` method takes into accout the current browser mode. The browsing sheet and style sheet that are used for the transformation can be set explicitly with the `setBrowsingSheet` and `setStyleSheet` methods respectively. Similarly, the browser mode for the current document can be set with the `setMode` method.

In future, alternative sheet languages may also be supported, for example CSS. Then it may also be possible to use XSLT, CSS and other sheets simultaneously.

In future, the transformation process may cover more steps than now. For example, the style sheet may be split up into separate style and layout sheets.

In future, it may also be possible to set the linkbase explicitly as well. The reason why it isn't already possible is that the XLink processor used for the transformation doesn't expose any method for setting linkbases explicitly, but it would be fairly easy to extend it in a way that this functionality is also provided.

The `DefaultSheetEngineImplementation` class relies heavily on the Cocoon XLink processor and the Xalan XSLT processor for performing the transformation process. Because the Cocoon XLink processor is meant to be used as a servlet, an `ArtificialServletEnvironment` is created which simulates a servlet environment.

An `ArtificialServletEnvironment` is a simulation of a servlet environment. This environment is necessary for the Cocoon XLink processor, which is meant to be used as a servlet, to be able to be used from within a standard Java application as well. Because this class is meant to be used only for this special purpose, only those methods called by the Cocoon XLink processor are implemented, all other methods throw an error.

The `setURI` method of the `ArtificialServletEnvironment` class sets the URI of the document that is processed by the XLink processor. Subsequent calls to the methods that don't throw an error will then return the various parts of the URI.

The `DefaultSheetEngineImplementation` keeps a representation of the parsed browsing sheet and the parsed style sheet, respectively, in memory to improve

the speed of the transformation.

4.3.5 Actions

The `de.uni_muenchen.informatik.pms.dt_browser.action` package is the home for classes and interfaces that represent the concept of actions as well as all actions of the current implementation.

The user can interact with the browser by invoking an action. Actions include, for example, opening a document, setting a style sheet or exiting the browser. Each action is represented by an implementation of one of the `BrowserAction`, `CloseAction` or `BrowserLinkListener` interfaces.

Actions are the link between the application logic and the user interface. Each action is represented by one or more user interface elements. The connection between actions and user interface is created by the `ActionFactoryListener` interface. The set of actions that make up the browser, as well as the structure of the actions, is defined by an XML file. This file is used by the `ActionFactory` class to instantiate the actions and to pass the information to the user interface, which will create UI elements for each of the actions.

The `BrowserAction` Interface

All actions exposed by the browser must be an implementation of `BrowserAction`.

As interfaces cannot define constructors such as

```
public BrowserAction(Browser browser);
```

each `BrowserAction` must implement the `setBrowser` method which associates the action with a browser.

One action has to be an implementation of the `CloseAction` interface, which is a subinterface of `BrowserAction`. This special action will be invoked when the browser is closed. The reason why closing the browser cannot be represented by a normal `BrowserAction` is that in a window user interface, for example, there is a special button for closing the window and therefore the browser which is supplied by the operating system. This special button is not part of the browser user interface, but its activation has to be handled by the browser, though.

The `BrowserLinkListener` Interface

A `BrowserLinkListener` is responsible for handling hyperlink events. Each time the user initiates the traversal of a hyperlink, the `hyperlinkUpdate` method will be called.

To associate a `BrowserLinkListener` with a browser instance, the `setBrowser` method is used in the same way as with the `BrowserAction` interface.

The `DefaultBrowserLinkListenerImplementation` is only able to handle simple hyperlinks, that is binary, unidirectional hyperlinks. This means that all

further hyperlink handling, for example n-ary and bidirectional hyperlinks, has to be done elsewhere.

In future, the `BrowserLinkListener` interface will be extended to a fully functional *hyperlink engine* (analogous to a sheet engine) which provides support for all XLink features and for those features that have been described in section 3.1.

The `ActionFactory` Class

The `ActionFactory` class is a factory class for vending objects from the `de.uni_muenchen.informatik.pms.dt_browser.action` package.

This class is a bit different from the other factory classes. It has no `getAction` method which directly returns a certain browser action, but it writes all actions of the *DT Browser* implementation to an `ActionFactoryListener` interface using the `writeActions` method.

The set of actions that a browser instance provides is specified in the XML file `browseractions.xml`. With this file, it is also possible to define the structure of the actions. When the *DT Browser* is loaded, this file will be parsed and stored within the `ActionFactory` class. The actual parsing is done by an instance of the `SAXHandler` class, which converts SAX events to `ActionFactoryListener` events.

An `ActionFactoryListener`, superinterface of `UI`, is used to register browser actions. During the parsing process of the `browseractions.xml` file, each parsed action declaration is be reported to the `ActionFactoryListener` which is then responsible for handling these events properly.

Currently, the following elements may be specified within the `browseractions.xml` file:

- `action`: Declares a normal `BrowserAction`.
- `closeaction`: Declares the browser's `CloseAction`. Each browser must have exactly one `CloseAction`.
- `group`: Browser actions may be bundled into groups. Actions with similar functionalities should be put into one group.
- `subgroup`: Groups may be further divided into subgroups. For information about how the default user interface renders groups and subgroups, see below.
- `linklistener`: Declares the browser's `BrowserLinkListener`. A browser must have exactly one `BrowserLinkListener`. The placing of this element does not interfere with the browser actions' declarations and their grouping, as a `BrowserLinkListener` is not a direct part of the user interface.

The formal XML document type definition (DTD) of the `browseractions.xml` file looks like this:

```
<!DOCTYPE browseractions [
<!ELEMENT browseractions (linklistener,group+)>
<!ELEMENT group (action|closeaction|subgroup)+>
<!ATTLIST group name CDATA #REQUIRED>
<!ELEMENT linklistener (#PCDATA)>
<!ELEMENT action (#PCDATA)>
<!ELEMENT closeaction (#PCDATA)>
<!ELEMENT subgroup EMPTY>
]>
```

Actions of the *DT Browser* Implementation

New Browser Creates a new browser instance. With the default user interface, this action opens a new browser window.

Close Browser Destroys the current browser instance. With the default user interface, this action closes the browser window. If only one browser instance is open, this action exits the *DT Browser* application.

Exit Browser Destroys all browsers instances and exits the *DT Browser* application.

Open Document from File Prompts the user for an XML source document from the local file system and opens it in the browser. The source document is transformed by the sheet engine to a presentation document, which is rendered by the rendering engine. The sheet engine takes into account all sheets referenced by the source document.

Open Document from URI Similar to above, but prompts the user for a URI instead of a local file.

Set Browser Mode Prompts the user for a browser mode and displays the current document with that mode. The set of modes the user can choose from is stems from the sheet declarations in the source document. The document currently being displayed is transformed by the sheet engine to a presentation document, which is rendered by the rendering engine. The sheet engine uses the sheets that belong to the selected mode.

Open Browsing Sheet from File Prompts the user for a browsing sheet from the local file system and applies it to the document currently being displayed. After updating the sheet engine, the source document is transformed and rendered using the current sheet and mode settings.

Open Browsing Sheet from URI Similar to above, but prompts the user for a URI instead of a local file.

Open Style Sheet from File Similar to above, but prompts the user for a style sheet from the local file system instead of a browsing sheet.

Open Style Sheet from URI Similar to above, but prompts the user for a URI instead of a local file.

New Reader's View Opens a new reader's view document, which is empty. As currently each browser is associated with exactly one reader's view, this has the effect of clearing the browser's reader's view.

Open Reader's View Prompts the user for a reader's view document from the local file system which is opened in the browser. The reader's view document is transformed by the sheet engine to a presentation document, which is rendered by the rendering engine. The sheet engine takes into account all sheets referenced by the reader's view document.

Save Reader's View Prompts the user for a file from the local file system in which the browser's reader's view document will be stored. This action stores the reader's view's elements as well as all its sheet settings.

Add Selected Node to Reader's View Adds the currently selected node of the source document tree to the reader's view tree. The node is be copied and added as the last child node of the currently selected node the reader's view tree. The reader's view document is transformed and rendered using the current sheet settings.

Remove Selected Node from Reader's View Removes the currently selected node of the reader's view tree and all its children. The reader's view document is transformed and rendered using the current sheet settings.

Create New Node Prompts the user for a line of text and adds it as a newly created node to the reader's view tree. The node will be inserted as the last child of the currently selected node of the reader's view tree. The reader's view document is transformed and rendered using the current sheet settings.

Open Reader's View Browsing Sheet from File Prompts the user for a browsing sheet from the local file system and applies it to the reader's view document. After updating the sheet engine, the reader's view document is transformed and rendered using the current sheet and mode settings.

Open Reader's View Browsing Sheet from URI Similar to above, but prompts the user for a URI instead of a local file.

Open Reader's View Style Sheet from File Similar to above, but prompts the user for a style sheet from the local file system instead of a browsing sheet.

Open Reader's View Style Sheet from URI Similar to above, but prompts the user for a URI instead of a local file.

4.3.6 Messages

The `de.uni_muenchen.informatik.pms.dt_browser.message` package is the home for classes and interfaces that provide the browser with means of displaying messages to the user. For several reasons, this package is separated from the `de.uni_muenchen.informatik.pms.dt_browser.ui` package.

In contrast to the `de.uni_muenchen.informatik.pms.dt_browser.ui` package, which is responsible for displaying the *normal* user interface, for example the presentation of the document, the reader's view etc., this package is meant to display messages that indicate some *abnormal* state of the browser, for example messages about errors or diagnostic messages.

In future, it should be possible to choose between several different styles or modes of displaying messages. In a *user mode*, only severe errors such as "Unable to open XML document" should be displayed, without any low-level details. An *expert mode*, however, should also display these details, for example "XML document is not valid". In addition to this, diagnostic messages, which are not errors, for example "XML document references linkbase X" will be displayed. This could be achieved by switching the current `MessageHandler` to another mode, or by replacing it with another `MessageHandler` implementation. The latter method is chosen here, because it gives greater flexibility to new implementors.

It should also be possible to change the message user interface independent from the browser's *normal* user interface. For example, while using a standard window user interface, the user may want to have a vast amount of diagnostic messages to be displayed on the standard ouput instead of a message window. Then he can redirect the output into a file and do further diagnostic processing afterwards.

In future, the message user interface might be changed while the browser is running, similar to the browser's *normal* user interface. So far, none of the above mentioned features is implemented. The current `MessageHandler` implementation simply writes all messages to the standard output.

The `MessageHandler` Interface

A `MessageHandler` is capable of displaying messages such as errors and diagnostic messages to the user. This interface defines the methods any message handler must implement. Each browser is associated with exactly one message handler.

Currently, the only method supported by this interface displays an exception to the user. In future, more methods may be added, for example for displaying errors, warnings, diagnostic messages etc.

Currently, each browser has its own message handler, but in future, it may be possible to share message handlers between browsers. For instance, all browsers

may use only one message handler which then displays all messages. The actual assignment of message handlers to browsers may be subject to user preferences.

The `DefaultMessageHandlerImplementation` class simply displays information about exceptions to the user.

Currently, the exceptions are just print out to the standard output. In future, a more sophisticated output may be implemented. This output may make use of an abstract user interface, which may be part of the `UI` interface or part of the `MessageHandler` interface itself.

4.3.7 User Interface

The `de.uni_muenchen.informatik.pms.dt_browser.ui` package is the home for classes and interfaces that represent the concept of a user interface, which is responsible for displaying the browser to the user.

The user interface is completely separated from the application logic, that is, the application logic classes and interfaces have no knowledge about the existence of a user interface and vice versa. Actions are the only place where application logic and user interface are connected.

The user interface is split up into two parts: `LoadingUI` and `UI`. Whereas the `LoadingUI` is only responsible for displaying some kind of information to the user while the browser is still loading, the `UI` displays the browser itself. The only reason for this split is to minimize the number of classes that must be loaded by the Java runtime system before it is able to display a message like "DT Browser is loading..." to the user.

The `UI` Interface

A `UI` object represents the browser's user interface. This interface defines the methods any user interface must implement. Each browser is associated with exactly one user interface, which displays the browser to the user and returns user input to the browser.

A newly created user interface is still invisible. To make is visible, call the `showBrowser` method. If the user wants to close the browser, a call to the `hideBrowser` method will make it invisible again.

The various `setXXX` methods will display something to the user, such as an XML presentation document or the document tree. Through the `getXXX` methods the browser can request information from the user, such as a filename, a URI etc. There is also a set of methods to dynamically change the reader's view, which is an integral part of the browser.

The `DefaultWindowUIHandlerImplementation` class represents a window user interface. This implementation relies heavily on the Java AWT classes and makes use of a `ClosingWindowListener` instance.

It presents each `BrowserAction` twice: Through a menu item and a toolbar button. In the menu bar, each action group is presented as a single menu, subgroups are be separated by menu separators. In the toolbar, action groups are separated by toolbar separators, there is no representation of subgroups. The single

`CloseAction` is presented like a normal browser action, but it will also be associated with the browser's `ClosingWindowListener`. The `BrowserLinkListener` is registered with the rendering engine to be able to report hyperlink traversal events.

A `ClosingWindowListener` waits for the browser window to be closed by the user. If a `windowClosing` event is received, the `CloseAction` associated with the browser will be called.

Currently, the `DefaultWindowUIImplementation` displays presentation document, source document tree, reader's view presentation document and reader's view tree all in one window. This is done for the ease of implementation, not for the comfort of the user and may be improved in the future.

The `LoadingUI` Interface

A `LoadingUI` object is able to display information to the user while the browser is still loading. This interface defines the methods any loading user interface must implement. Note that the loading user interface is meant to be displayed only when the browser is loaded first and not when a new browser window is opened, for example.

A newly created loading user interface is still invisible. To make is visible, call the `show` method. To make it invisible again, call the `hide` method.

The `DefaultWindowLoadingUI` class displays an image to the user. This implementation relies heavily on the Java AWT classes.

4.4 Implementation Details

This section describes details of the implementation, such as the runtime environment, the file and directory structure, the third party software and the tools included with the *DT Browser* implementation.

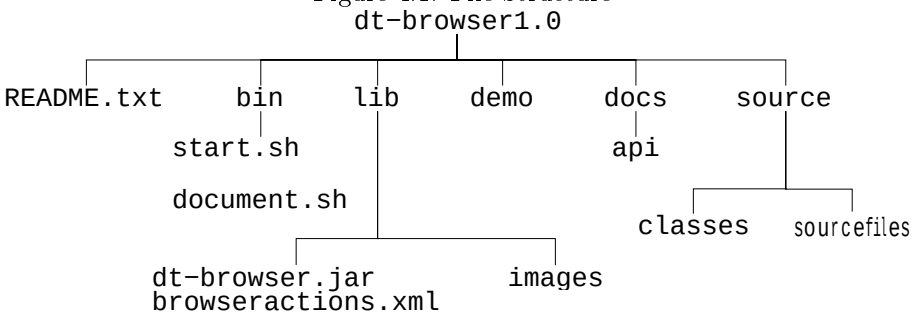
4.4.1 Implementation Language

The browser toolkit is implemented in Java. For a good introduction as well as more information about the Java language, see *The Java Tutorial* [CW98]. Java was chosen for several reasons:

First, it is a modern language which is easy to use. Java allows to develop large software systems like the browser toolkit. The Java language is object-oriented and offers features like packages and exception handling.

Second, a lot of tools than can be used to implement the browser toolkit, for example XML parsers, XSLT processors and also the standard Java API, of course, are available in Java. The Java API provides things like networking, a window toolkit and container classes, for example. It is one of the goals of the browser toolkit not to implement everything on its own, but to use existing standards and tools as far as possible, whenever it is reasonable.

Figure 4.1: File Structure



Finally, Java is platform independent and most likely, as far as one can see, will not vanish in the near future. This is important because the browser toolkit is also meant not to vanish in the near future, but to be used for experimenting with new browsing functionalities, which may outlive specific operating systems or hardware platforms.

The browser toolkit was developed under SuSE Linux 7.0 and the Java 2 SDK, Standard Edition, v 1.3. In order to start the browser (no compilation or documentation), only the Java 2 Runtime Environment, Standard Edition, v 1.3 (JRE) is needed, however. For more information about Java software, see the *Java Software Home Page* [Jav].

4.4.2 File Structure

This section describes the files and directories that make up the *DT Browser* implementation. Figure 4.1 shows the most important files and directories.

Assuming that the browser is installed at `/dt-browser1.0`, here is a description of all files and directories:

`/dt-browser1.0`

The root directory of the *DT Browser* implementation. Contains the file `README.txt` with some first aid for the novice user.

`/dt-browser1.0/bin`

Executable shell scripts for starting (`start.sh`), compiling (`compile.sh`) and documenting (`document.sh`) the browser. Also contains helper files for these scripts. Currently, only scripts for Unix-like operating systems are available, but in future, scripts for other operating systems such as MS Windows may be added.

`/dt-browser1.0/lib`

Files needed to start the browser:

- `dt-browser.jar`: contains all the class files of the implementation
- `browseractions.xml`: user interface configuration file, that is the description of all actions exposed by the browser
- `emptydocument.xml`, `nopsheet.xsl`: dummy XML source document and XSLT stylesheet which are used when no source document or sheet is loaded into the browser, respectively.

`/dt-browser1.0/lib/images`

Image files used by the browser. Includes the image shown while loading the browser and icons for all browser actions.

`/dt-browser1.0/demos`

XML source documents, sheets and linkbases for demonstrating the features of the browser.

`/dt-browser1.0/docs`

Documentation for the browser. The root file of the documentation is `index.html`.

`/dt-browser1.0/docs/api`

API documentation for the browser. The root file of the documentation is `index.html`.

`/dt-browser1.0/source`

Java source code and Javadoc source files of the browser. Currently, the source code is contained in the two directories `com` and `de`.

`/dt-browser1.0/source/classes`

Class files of the browser, that is, the compiled source code.

4.4.3 Third Party Software

The implementation of the browser toolkit relies heavily on third party modules. Almost all of them stem from the Apache XML Project [APA]. This has two advantages: First, the risk of incompatibilities is much smaller than if modules from a variety of parties were used. Second, software developed and made available by a large organization like Apache is likely to exist for a long time, as opposed to software provided by small companies or even private persons.

Java 2 SDK

Not really a third party module, but it contributes to the browser toolkit as well. The Java 2 Platform provides things like container classes, networking and an abstract window toolkit. Especially, the EditorPane class is used as rendering engine for the browser, that is, it displays the presentation documents which are represented in HTML.

Xerces

An XML parser implementing the W3C XML and DOM (Level 1 and 2) standards, as well as the defacto SAX (version 2) standard. The parser is highly modular and configurable. Initial support for XML Schema (draft W3C standard) is also provided. Xerces is used to read all XML files the browser processes, that is the XML source doucments, sheets, linkbases as well as the user interface configuration file.

Xalan

Xalan is an XSLT processor for transforming XML documents into HTML, text, or other XML document types. Xalan represents a complete and robust reference implementation of the W3C Recommendations for XSL Transformations (XSLT) and the XML Path Language (XPath). It is used to process an XML source document with a browsing sheet or a style sheet.

Cocoon XLink Processor

The XLink processor is mainly concerned with interpreting Extended XLinks and converting them to Simple XLinks. It supports a large part of the XLink specification and a part of the XPointer specification. The extensions to XPath that XPointer defines are not supported, thus the possibility to address on a sub-element level is not supported. The Cocoon XLink Processor is used to process an XML source document with one or more linkbases.

SAX

SAX, the Simple API for XML, is a standard interface for event-based XML parsing. It is used to process the user interface configuration file of the browser toolkit.

Although the following two modules are not used by the current *DT Browser* implementation, they represent interesting alternative approaches for several parts of the browser and are likely to be used in future versions:

SAXON

SAXON includes a Java library, which supports a processing model similar to the one of XSL, but allows full programming capability, which you need if you

want to perform complex processing of the data or to access external services such as a relational database. SAXON might be used for processing XML source documents with sheets and linkbases.

FOP

FOP is a print formatter driven by XSL formatting objects. It is a Java 1.1 application that reads a formatting object tree and then turns it into a PDF document. It also contains an XSL FO rendering engine, which might replace or coexist with the HTML rendering engine currently used by the browser toolkit.

Cocoon

Cocoon is a publishing framework that relies on new W3C technologies (such as DOM, XML, and XSL) to provide web content. Although not being a browser, it has many similarities with the browser toolkit, for example a separation of several layers like content, style and logic. Therefore it might provide some features that will enter the browser toolkit in future as well.

4.4.4 Tools of the *DT Browser* Implementation

The *DT Browser* implementation comes with a set of tools for starting, compiling and documenting the browser. Currently, only tools for Unix-like operating systems are available, but in future, tools for other operating systems such as MS Windows may be added as well. As these tools are simple shell scripts which do contain nothing more than Java SDK tool invocations, it is not much work to generate MS Windows version of these, for example. All of these tools are based on the tools that come with the Java SDK.

Starting the Browser

The *start* tool (`start.sh` for Unix-like systems) starts the browser. It is simply an invocation of the `CommandLine` class using the `java` tool of the Java SDK or the Java Runtime Environment (JRE).

As explained in subsection 4.2.2, it is possible to specify interface implementation that will be used by the browser through system properties. As the `java` tool allows to set system property values using the `D` option, the desired setting can be specified within the start tool. Apart from the system properties explained in the factory subsection, it is necessary to set the value of the SAX property `org.xml.sax.driver` to the class name of an XML parser with SAX interface. This parser will be used to process the user interface configuration file `browseractions.xml`.

All files needed to start the browser reside in the `lib` directory. This means that all other files and directories of the *DT Browser* implementation, for example `demo`, `docs` and `source`, are not needed to start the browser. To create a runnable distribution of the browser, without source code, demos and documentation, only the `lib` directory and the start tool is needed.

As stated above, the `lib` directory contains the `dt-browser.jar` file. Currently, only class files are included within this jar file, all other files needed to start the browser, for example `browseractions.xml` or the `image` directory reside outside the jar file in the `lib` directory. In future, however, all of these files shall be included within the jar file, so that it will be the only file needed to start the browser (apart from the start tool).

Compiling the Browser

The *compile* tool performs two tasks: First, it compiles the source code to class files and second, it packs all class files into a jar file. It uses the Java SDK `javac` and `jar` tools, respectively.

The source code files reside in the `source` directory. The class files reside in the `source/classes` directory. They are only intermediate files, because the `dt-browser.jar` jar file is used to invoke the browser, therefore the `classes` directory resides in the `source` directory.

The `javac` tool uses the `sourcefiles.txt` file for compiling all classes and interfaces. This file contains the names of all source code files that have to be compiled. If a new class or interface is added to the *DT Browser* implementation, then this file has to be updated as well.

The `dt-browser.jar` jar file currently contains only class files (see above).

Documenting the Browser

The *document* tool generates the API documentation for the *DT Browser* using the Java SDK `javadoc` tool.

The API documentation is generated from so-called *doc comments* which are part of the source code. Additional documentation files such as `overview.html` are also contained within the `source` directory. The documentation is generated within the `docs/api` directory.

The `javadocc` tool uses the `packages.txt` file for documenting all packages. This file contains the names of all packages of the *DT Browser* implementation. If a new packages is added to the implementation, then this file has to be updated as well.

The target group of the documentation are developers who extend the *DT Browser* with new functionality or otherwise develop it further. This means that these developers usually extend and modify the existing source code. It does neither primarily address to those who want to use the *DT Browser* within their own applications, nor is it an API specification. This leads to several consequences:

Above all, the documentation is called *API documentation* instead of *API specification*. All classes and members are included in the documentation, that is private classes and members as well, because the documentation is meant to be used at the same time as looking at the source code while extending or modifying it. Also, information about version and author is included, which normally would not be found in an API specification.

In addition to giving information about boundary conditions, argument ranges and corner cases, the documentation also defines common programming terms, gives conceptual overviews and includes examples for developers. Because the documentation is primarily meant to be used by people who develop the *DT Browser* further, it contains also information about known weaknesses of the API design and the implementation as well as known bugs of the implementation. Furthermore, information about possible future enhancements both of the API as well as the implementation is provided.

Classpath Settings

All tools that have been described above work with the concept of a *classpath*. This classpath has to contain all jar files that are needed to run the tool. As the *DT Browser* implementation relies heavily on third party modules, those modules' jar files have to be parts of the class path as well.

The classpath is similar for all tools described above, with one exception. The jar files of all third party modules are always part of the classpath, as they are needed to run the tool. The exception is the start tool. It also needs the *DT Browser* jar file to start the browser, of course. The other tools can run without this jar file, as they are generating their output from the source code of the browser toolkit.

Bibliography

[ABC+00] S. Adler, A. Berglund, J. Caruso, S. Deach, P. Grosso, E. Gutentag, A. Milowski, S. Parnell, J. Richman, and S. Zilles. Extensible Stylesheet Language (XSL) Version 1.0. W3C Working Draft, October 2000. See http://www.w3.org/TR/xsl/.

[ABS00] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web – From Relations to Semistructured Data and XML.* Morgan Kaufmann, 2000. ISBN 1-55860-622-X.

[ACK+00] V. Apparao, M. Champion, J. Kesselman, J. Robie, and P. Sharpe. Document Object Model (DOM) Level 2 Traversal and Range Specification Version 1.0. W3C Proposed Recommendation, September 2000. See http://www.w3.org/TR/DOM-Level-2-Traversal-Range.

[AHW00] V. Apparao, P. Le Hégaret, and C. Wilson. Document Object Model (DOM) Level 2 Style Specification Version 1.0. W3C Proposed Recommendation, September 2000. See http://www.w3.org/TR/DOM-Level-2-Style.

[APA] The Apache XML Project. See http://xml.apache.org/.

[BB99] J. Bosak and T. Bray. XML and the Second-Generation Web. *Scientific American*, 1999. See http://www.sciam.com/1999/0599issue/0599bosak.html.

[BE00] F. Bry and N. Eisinger. Data modeling with markup languages. See http://www.pms.informatik.uni-muenchen.de/forschung/datamodeling-markup.html, July 2000.

[BEG99] F. Bry, N. Eisinger, and T. Geisler. Hauptseminar Elektronische Medien in der Lehre: Modellierungsaspekte, 1999. See http://www.pms.informatik.uni-muenchen.de/lehre/seminar/emedien/99ws00/.

[BG00] D. Brickley and R.V. Guha. Resource Description Framework (RDF) Schema Specification 1.0. W3C Candidate Recommendation, March 2000. See http://www.w3.org/TR/rdf-schema.

[BHL99] T. Bray, D. Hollander, and A. Layman. Namespaces in XML. World Wide Web Consortium, January 1999. See http://www.w3.org/TR/REC-xml-names.

[BLLJ98] B. Bos, H. W. Lie, C. Lilley, and I. Jacobs. Cascading Style Sheets, level 2. W3C Recommendation, May 1998. See http://www.w3.org/TR/REC-CSS2.

[Bos97] J. Bosak. XML, Java, and the future of the Web. Technical report, Sun Microsystems, 1997. See http://sunsite.unc.edu/pub/sun-info/standards/xml/why/xmlapps.htm.

[Bou99] R. Bourret. XML and Databases. See http://www.rpbourret.com/xml/XMLAndDatabases.htm, September 1999.

[BPSM00] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0 (Second Edition). W3C Recommendation, October 2000. See http://www.w3.org/TR/REC-xml.

[Bra98] N. Bradley. *The XML Companion.* Addison Wesley, 1998. ISBN 0-201-342855.

[Bun97] P. Buneman. Semistructured Data. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 117–121, 1997.

[CD99] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0. W3C Recommendation, November 1999. See http://www.w3.org/TR/xpath.

[CH00] L. Cable and A. Le Hors. Document Object Model (DOM) Level 2 Views Specification Version 1.0. W3C Proposed Recommendation, September 2000. See http://www.w3.org/TR/DOM-Level-2-Views.

[Cha99] P. Chan. *The Java Developers Almanac.* The Java Series. Addison-Wesley, 1999.

[Cla99a] J. Clark. Associating Style Sheets with XML documents Version 1.0. W3C Recommendation, June 1999. See http://www.w3.org/TR/xml-stylesheet.

[Cla99b] J. Clark. XSL Transformations (XSLT) Version 1.0. W3C Recommendation, November 1999. See http://www.w3.org/TR/xslt.

[Cow00] J. Cowan. XML Information Set. W3C Working Draft, July 2000. See http://www.w3.org/TR/xml-infoset.

[CW98] M. Campione and K. Walrath. *The Java Tutorial Second Edition: Object-Oriented Programming for the Internet.* The Java Series. Addison-Wesley, 1998.

[CW99] M. Campione and K. Walrath. *The JFC Swing Tutorial: A Guide to Constructing GUIs.* The Java Series. Addison-Wesley, 1999.

[CWH98] M. Campione, K. Walrath, and A. Huml. *The Java Tutorial Continued: The Rest of the JDK.* The Java Series. Addison-Wesley, 1998.

[dBH94] P. de Bra and G. Houben. A Formal Approach to Analyzing the Browsing Semantics of Hypertext. In *Proceedings of Computation and Neural Systems (CNS94)*, Monterey, CA, July 1994.

[dBHK92] P. de Bra, G. Houben, and Y. Kornatzky. An Extensible Data Model for Hyperdocuments. In *4th ACM Conference on Hypertext*, pages 222–231, Milan, 1992.

[DHH+00] M. Davis, A. Le Hors, P. Le Hégaret, J. Robie, and L. Wood. Document Object Model (DOM) Level 2 Core Specification Version 1.0. W3C Proposed Recommendation, September 2000. See http://www.w3.org/TR/DOM-Level-2-Core.

[DMOT00] S. DeRose, E. Maler, D. Orchard, and B. Trafford. XML Linking Language (XLink) Version 1.0. W3C Candidate Recommendation, July 2000. See http://www.w3.org/TR/xlink/.

[ECM99] Standard ECMA-262. ECMAScript Language Specification, 1999. See ftp://ftp.ecma.ch/ecma-st/Ecma-262.pdf.

[FLM98] D. Florescu, A. Levy, and A. Mendelzon. Database Techniques for the World-Wide Web: A Survey. In *SIGMOD Record, 27(3)*, pages 59–74, 1998.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.

[GP00] C. F. Goldfarb and P. Prescod. *The XML Handbook*. Prentice Hall PTR, 2000. ISBN 0-13-014714-1.

[Har99] E. R. Harold. *XML Bible*. IDG Books, 1999.

[HH00] A. Le Hors and P. Le Hégaret. Document Object Model (DOM) Level 2 HTML Specification Version 1.0. W3C Proposed Recommendation, September 2000. See http://www.w3.org/TR/DOM-Level-2-HTML.

[HS90] F. Halasz and M. Schwartz. The Dexter Hypertext Refernce Model. In *Proceedings of the Hypertext Workshop*, pages 95–133, Gaithersburg, Md, March 1990. National Institute of Standards and Technology. NIST Special Publication 500-178.

[ISO] ISO 10179:1996. Information technology — Processing lanuages — Document Style Semantics and Specification Language (DSSSL). See ftp://ftp.ornl.gov/pub/sgml/WG8/DSSSL/.

[ISO97] International Standard ISO/IEC 10744: Information technology – Hypermedia/Time-based Structuring Language (HyTime), August 1997. See http://www.ornl.gov/sgml/wg8/docs/n1920/html/n1920.html.

[Jav] Java Software Home Page. See http://www.javasoft.com/.

[JDM00] R. Daniel Jr., S. DeRose, and E. Maler. XML Pointer Language (XPointer) Version 1.0. W3C Candidate Recommendation, June 2000. See http://www.w3.org/TR/xptr.

[LB99] H. W. Lie and B. Bos. Cascading Style Sheets, level 1. W3C Recommendation, January 1999. See http://www.w3.org/TR/REC-CSS1.

[LS99] O. Lassila and R. R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. W3C Recommendation, February 1999. See http://www.w3.org/TR/REC-rdf-syntax.

[Man98a] F. Manola. Some Web Object Model Construction Technologies. Technical report, Object Services and Consulting, Inc., September 1998. See http://www.objs.com/OSA/wom-II.htm.

[Man98b] F. Manola. Towards a Web Object Model. Technical report, Object Services and Consulting, Inc. (OBJS), February 1998. See http://www.objs.com/OSA/wom.htm.

[Mar00] J. Marsh. XML Base. W3C Candidate Recommendation, September 2000. See http://www.w3.org/TR/xmlbase.

[MPD98] A. Michard and G. Pham-Dac. Description of Collections and Encyclopaedias on the Web using XML. In *Archives and Museum Informatics, 12*, pages 39–79. Kluwer Academic Publishers, 1998.

[Pem00] S. Pemberton. XHTML[tm] 1.0: The Extensible HyperText Markup Language. W3C Recommendation, January 2000. See http://www.w3.org/TR/xhtml1.

[Pix00] T. Pixley. Document Object Model (DOM) Level 2 Events Specification Version 1.0. W3C Proposed Recommendation, September 2000. See http://www.w3.org/TR/DOM-Level-2-Events.

[Rag97] D. Raggett. HTML 3.2 Reference Specification. W3C Recommendation, January 1997. See http://www.w3.org/TR/REC-html32.

[RHJ99] D. Raggett, A. Le Hors, and I. Jacobs. HTML 4.01 Specification. W3C Recommendation, December 1999. See http://www.w3.org/TR/html401.

[vH94] E. van Herwijnen. *Practical SGML*. Kluwer Academic Publishers, 1994. ISBN 0-7923-9434-8.

[W3C99] W3C. XML in 10 points. See http://www.w3.org/XML/1999/XML-in-10-points, 1999.

[Woo98] L. Wood. Document Object Model (DOM) Level 1 Specification Version 1.0. W3C Recommendation, October 1998. See http://www.w3.org/TR/REC-DOM-Level-1.